# ALGORYTMY
# W PRZYKŁADACH

Tekst został opracowany na podstawie zasobów internetowych (m.in. teksty mgr Jerzego Wałaszka) , podręcznika „Informatyka dla LO” WSIP , „ Algorytmy + Struktury danych = Programy” N. Wirtha , „Algorytmy” M.M.Sysło

Opracowanie : Dariusz Nyk

## SPIS TREŚCI

**Wstęp.** 4
Schemat blokowy 5
Instrukcja iteracji 8
Złożoność obliczeniowa algorytmu. 10
**1. Wyznaczanie NWD i NWW – algorytm Euklidesa** 13
**2. Największy i najmniejszy element zbioru.** 16
**3. Poszukiwanie lidera w zbiorze.** 17
**4. Przeszukiwanie sekwencyjne.** 20
**5. Wyszukiwanie z wartownikiem.** 21
**6. Wyszukiwanie najczęstszego elementu zbioru.** 24
6.1. Drugi największy element zbioru. 32
6.2. K-ty największy element zbioru. 34
6.3. K-ty największy element zbioru – wyszukiwanie szybkie . 36
**7. Sito Erastotenesa.** 41
**8. Sortowanie zbioru.** 49
8.1. Sortowanie zwariowane (Bogo Sort) 52
8.2. Sortowanie naiwne (głupie). 57
8.3. Sortowanie bąbelkowe. 58
8.4. Sortowanie przez wybór. 65
8.5. Sortowanie przez wstawianie. 67
8.6. Sortowanie metodą Shella. 71
8.7. Rekurencja. 75
8.8. Sortowanie przez scalanie. 76
8.9. Sortowanie stogowe (przez kopcowanie , heap sort) 83
*8.9.1. Drzewo binarne.* 83
*8.9.2. Tworzenie kopca* 85
*8.9.3. Rozbiór kopca.* 88
8.10. Sortowanie szybkie. 92
8.11. Sortowanie dystrybucyjne. 97
*8.11.1. Sortowanie rozrzutowe.* 97
*8.11.2. Sortowanie kubełkowe.* 102
*8.11.3. Sortowanie przez zliczanie.* 106
8.12. Podsumowanie. 112
**9. Binarne kodowanie liczb.** 113
9.1. System dziesiętny. 113
9.2. Schemat Hornera. 116
9.3. Schemat Hornera jako sposób obliczania wartości liczby. 117
9.4. Przeliczanie liczb systemu dziesiętnego na dowolny system 119
9.5. Liczby zmiennoprzecinkowe. 121
**10. System binarny.** 128
10.1. BIT – podstawowa jednostka informacji. 128
10.2. Kod binarny. 129
10.3.Zastosowanie kodów binarnych. 130
*10.3.1. Kodowanie grafiki.* 130
*10.3.2. Kodowanie znaków.* 131
10.3.Bajt. 132
10.4. Mnożniki binarne. 132
**11. Naturalny system dwójkowy.** 133
11.1. Wartość liczby w naturalnym systemie dwójkowym. 133

11.2. Zakres liczby binarnej........133
11.3. Schemat Hornera dla liczb binarnych........133
11.4. Przeliczanie liczb dziesiętnych na binarne........134
11.5. Dwójkowy system stałoprzecinkowy........135
*11.5.1. Wartość dwójkowej liczby stałoprzecinkowej........135*
*11.5.2. Zakres binarnych liczb stałoprzecinkowych........137*
11.6. Operacje arytmetyczne na systemie dwójkowym........137
*11.6.1. Dodawanie........137*
*11.6.2. Odejmowanie........139*
*11.6.3. Mnożenie........140*
*11.6.4. Dzielenie........141*
11.7. Szybkie potęgowanie liczb........142
**12. Kodowanie liczb binarnych ze znakiem........143**
12.1. Zapis „znak-moduł"........144
*12.1.1. Wartość dziesiętna liczby w zapisie ZM........144*
12.2. Zapis uzupełnień do 1 – U1 (1C - One's Complement)........145
*12.2.1. Przeliczanie liczb dziesiętnych na zapis U1........146*
*12.2.2. Stałoprzecinkowy zapis U1........148*
*12.2.3. Dodawanie w U1........148*
*12.2.4.Odejmowanie w U1........149*
12.3. Zapis uzupełnień do 2 – U2 (2C - Two's Complement )........150
*12.3.1. Liczba przeciwna do liczby w U2........151*
*12.3.2. Przeliczanie liczb dziesiętnych na zapis U2........152*
*12.3.3. Stałoprzecinkowy zapis U2........154*
*12.3.4. Dodawanie i odejmowanie w U2........155*
*12.3.5. Mnożenie w U2........155*
*12.3.6. Dzielenie w U2........156*
**13. Pozostałe kody binarne........156**
13.1. Kod BCD........156
13.2. Kod Gray'a........158
*13.2.1. Wyznaczanie i-tego wyrazu n-bitowego kodu Gray'a........159*
*13.2.2.Rekurencyjny algorytm tworzenia wyrazów kodu Gray'a........160*
**14. Szyfrowanie danych........163**
14.1. Steganografia........163
14.2. Kryptografia........163
14.3. Szyfrowanie przez przestawianie........166
14.4. Szyfrowanie przez podstawianie........168
14.5. Kryptografia z kluczem jawnym........177
*14.5.1. Szyfr RSA........179*
14.6. Potwierdzenie autentyczności i podpis elektroniczny........184

# ALGORYTMY W PRZYKŁADACH

## Wstęp.

Algorytm – skończony ciąg czynności, przekształcający zbiór danych wejściowych na zbiór danych wyjściowych (wyników).

Etapy konstruowania algorytm :

1) sformułowanie zadania – ustalamy jaki problem ma rozwiązywać algorytm
2) określenie danych wejściowych – ich typu ( w typie określamy, czy dane są liczbami rzeczywistymi, całkowitymi, czy znakami, czy też innego typu)
3) określenie wyniku oraz sposobu jego prezentacji
4) ustalenie metody wykonania zadania (może być kilka metod na rozwiązanie – wybieramy tą, która według nas jest najlepsza)
5) zapisanie algorytmu za pomocą wybranej metody (z punktu 4 )
6) Analiza poprawności rozwiązania
7) Testowanie rozwiązania dla różnych danych (algorytm musi być uniwersalny, aby służyć do rozwiązywania zadań dla różnych danych wejściowych)
8) Ocena skuteczności algorytmu ( praktyczna ocena algorytmu : np. szybkości, skomplikowania)

Sposoby zapisu algorytmu.

Do najczęściej używanych sposobów zapisu algorytmu należą :

1) lista kroków
2) pseudojęzyk (pseudokod)
3) graficzna prezentacja za pomocą schematu blokowego
4) zapis w danym języku programowania

Zadanie : znaleźć średnią arytmetyczną dwóch liczb rzeczywistych

**Ad. 1** Lista kroków charakteryzuje się tym, że każdy wiersz opisujący pojedynczy krok realizowanej czynności jest numerowany.

1) pobierz pierwszą liczbę
2) pobierz drugą liczbę
3) dodaj liczby do siebie
4) wynik dodawania podziel przez 2
5) wyświetl otrzymaną wartość
6) zakończ

**Ad. 2.** Pseudojęzyk jest metodą pośrednią między zapisem za pomocą listy kroków a zapisem w języku programowania.

- początek
- wprowadzenie x i y rzeczywistych
- wykonanie działania (x+y)/2
- pisz wynik
- koniec

**Ad. 4.** Ten problem zapisany w postaci programu w języku Turbo Pascal

```
Program Srednia;

Var x, y : Real;
Begin
        Readln (x);
        Readln (y);
        Writeln ('Średnia arytmetyczna wprowadzonych liczb wynosi : ' (x+y)/2 :7:2);
End.
```

Przedstawiony tu algorytm liczenia średniej jest wykonywany zawsze w tej samej kolejności, niezależnie od wartości danych wejściowych.

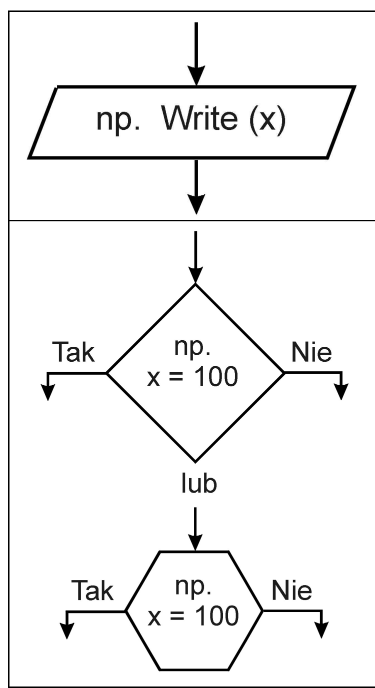
**Algorytm liniowy (sekwencyjny)** – algorytm, w którym kolejność wykonywanych czynności jest taka sama i niezależna od wartości danych wejściowych.

**Ad. 3.** Zapis za pomocą schematu blokowego.

### *Schemat blokowy* .

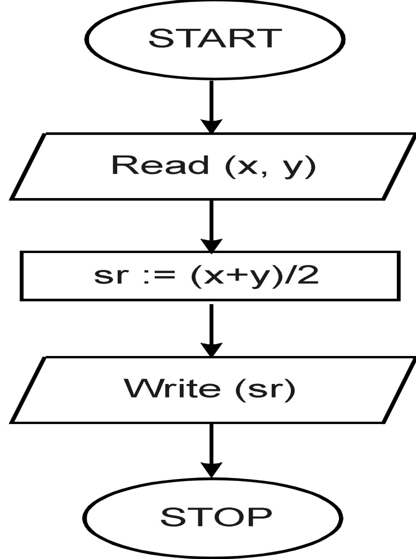
Schemat blokowy przedstawia algorytm w postaci symboli graficznych, podając szczegółowo wszystkie operacje arytmetyczne, logiczne, przesyłania, pomocnicze wraz z kolejnością ich wykonywania. Składa się on z wielu elementów, wśród których podstawowym jest blok.

| Wygląd bloku | Opis |
|---|---|
| START STOP | **Bloki graniczne** – początek i koniec algorytmu.<br>Mają kształt owalu. Z bloku Start wychodzi tylko jedno połączenie; każdy schemat blokowy musi mieć dokładnie jeden blok START.<br>Każdy schemat blokowy musi mieć co najmniej jeden blok STOP. |
| | **Łącznik** pomiędzy blokami – określa kierunek przepływu danych lub kolejność wykonywanych działań (ścieżka sterująca) |
| | **Blok kolekcyjny** – łączy kilka różnych dróg algorytmu |
| np. x:= 10 | **Blok operacyjny** – zawiera operację lub grupę operacji, w których wyniku ulega zmianie wartość zmiennej (tu : nadanie zmiennej x wartości 10).<br>Bloki operacyjne mają kształt prostokąta , wchodzi do niego jedno połączenie i wychodzi też jedno. |

<table>
<tr>
<td></td>
<td>Blok wejścia / wyjścia – blok odpowiedzialny za wykonanie operacji wprowadzania i wyprowadzania danych, wyników, komunikatów.<br>Ma kształt równoległoboku, wchodzi i wychodzi z niego jedno połączenie.</td>
</tr>
<tr>
<td></td>
<td>Blok decyzyjny – określa wybór jednej z dwóch możliwych dróg działania.<br>Ma kształt rombu lub sześciokąta. Wchodzi do niego jedno połączenie, a wychodzą dwa : TAK – gdy warunek wpisany wewnątrz jest spełniony oraz NIE – gdy warunek wpisany wewnątrz nie jest spełniony. Wybór kształtu bloku zależy od nas.</td>
</tr>
</table>

Istnieją oczywiście jeszcze inne bloki, stosowane przy dużych projektach, ale my z nich nie będziemy korzystać.

Zapiszmy teraz nasz algorytm liczenia średniej w postaci schematu blokowego :




Mamy już algorytm, ale brakuje nam jeszcze jednego czynnika sprawiającego by zadanie rozwiązywane za pomocą algorytmu było przedstawione w pełni precyzyjnie – specyfikacji problemu algorytmicznego.
**Specyfikacją problemu algorytmicznego** nazywamy dokładny opis problemu algorytmicznego, który ma zostać rozwiązany oraz podanie informacji o danych wejściowych i wyjściowych.
Czyli przed naszym algorytmem powinien znaleźć się dodatkowy zapis :
**Problem algorytmiczny** : obliczenie średniej arytmetycznej dwóch licz rzeczywistych
**Dane wejściowe** : $x , y \in R$
**Dane wyjściowe** : $sr \in R$ – średnia liczb x i y

Dane w algorytmie są najczęściej przedstawiane za pomocą liter lub nazw.
**Zmienną** nazywamy obiekt występujący w algorytmie, określony przez nazwę i służący do zapamiętywania określonych danych. Zmienna musi mieć określony typ (rzeczywisty, łańcuchowy itp.)

Jeśli stałe i zmienne połączymy operatorami to otrzymamy wyrażenie. Stałe i zmienne występujące w wyrażeniu nazywamy **operandami**.

| **Operatory arytmetyczne** | | **Operatory relacji** | |
|---|---|---|---|
| Symbol | Znaczenie | Symbol | Znaczenie |
| + | dodawanie | = | równy |
| - | odejmowanie | > | większy |
| * | mnożenie | >= | większy lub równy |
| / | dzielenie | < | mniejszy |
| div | dzielenie całkowite (3div2 = 1) | <= | mniejszy lub równy |
| mod | reszta z dzielenia liczb całkowitych | <> | różny |

Operator przypisania **:=**
<u>Po lewej stronie operatora przypisania może stać tylko zmienna !</u>

*Operatory logiczne*
and - koniunkcja (iloczyn zdań) /\
or - alternatywa (suma zdań) \/
not - negacja (zaprzeczenie zdania) ~

Rachunek zdań

| x | y | x and y (x/\y) | x or y (x\/y | x | not x (~x) |
|---|---|---|---|---|---|
| **True** | True | True | True | True | False |
| True | False | False | True | False | True |
| False | True | False | True | | |
| False | False | False | False | | |

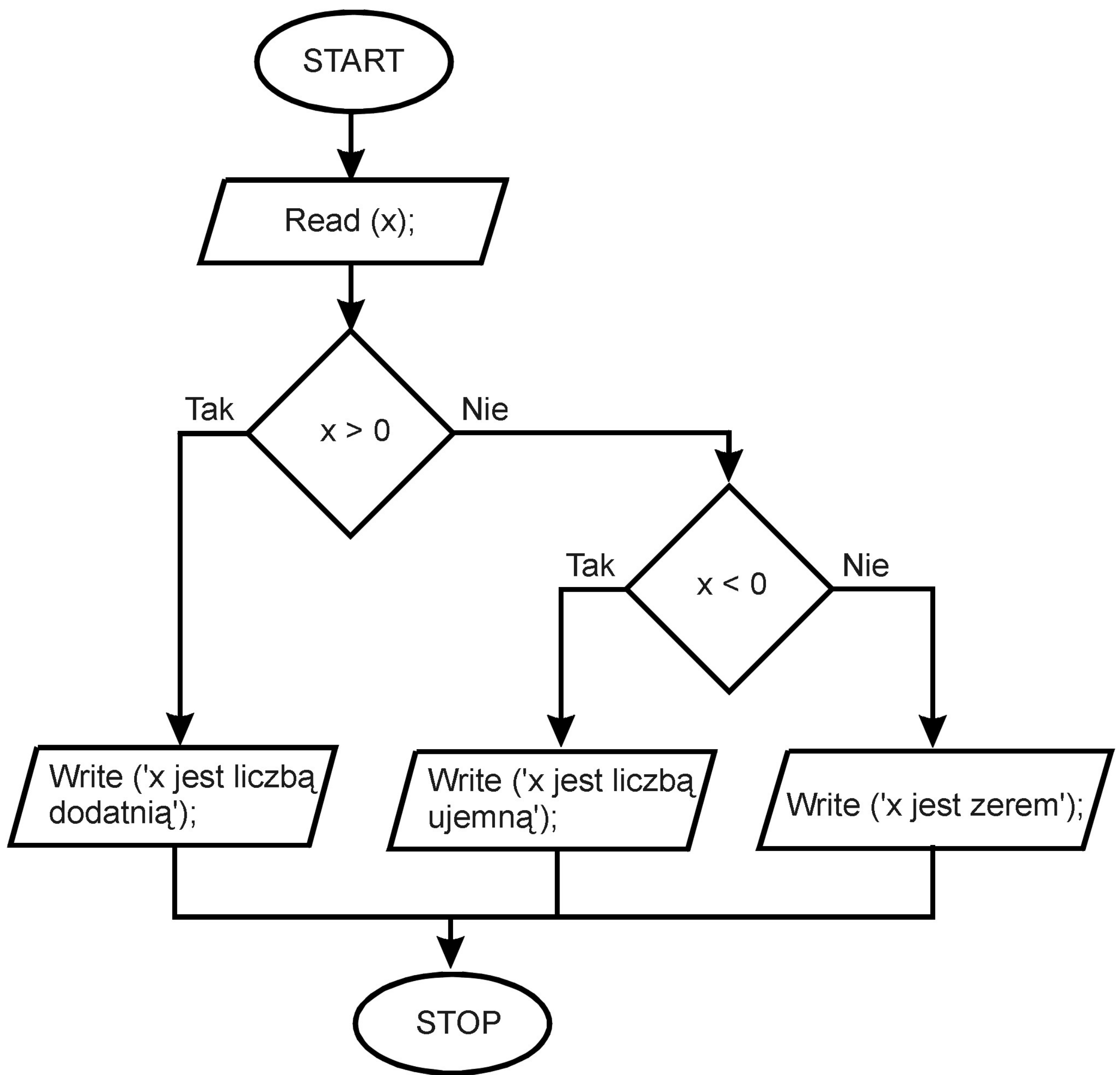
Do tej pory zajmowaliśmy się **algorytmami liniowymi (sekwencyjnymi)** – algorytm, w którym kolejność wykonywanych czynności jest taka sama i niezależna od wartości danych wejściowych.

Wprowadźmy sobie dodatkowo instrukcję wyboru (warunkową) oraz pojęcie algorytmu rozgałęzionego.

**Instrukcją wyboru (warunkową)** to instrukcja, której wykonanie jest uzależnione od tego czy dany warunek został spełniony czy też nie.

**Algorytm rozgałęziony** to algorytm, w którym występują instrukcje warunkowe.

Specyfikacja problemu algorytmicznego
Problem algorytmiczny : określenie znaku wprowadzonej liczby
Dane wejściowe : $x \in R$
Dane wyjściowe : wynik w postaci napisu „liczba x jest liczbą dodatnią” jeśli x>0 , „liczba x jest zerem” jeśli x=0 lub „liczba x jest ujemna” jeśli x<0.




Warunek musi być tak określony, aby jego ocena prawdziwości była jednoznaczna.

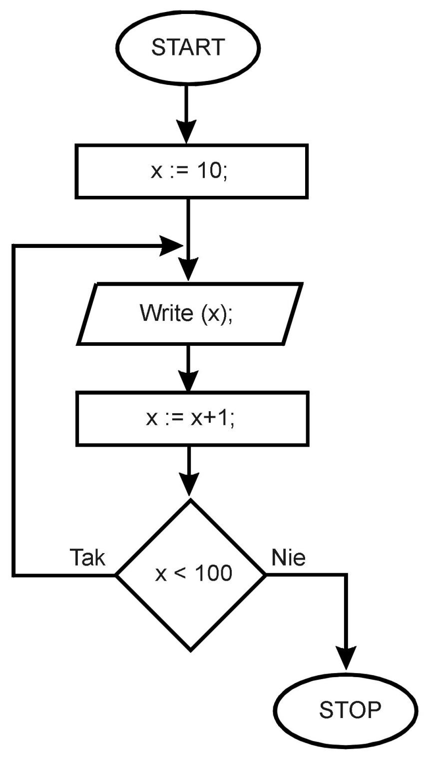
## *Instrukcja iteracji*

Zadanie : wypisać wszystkie liczby dwucyfrowe.

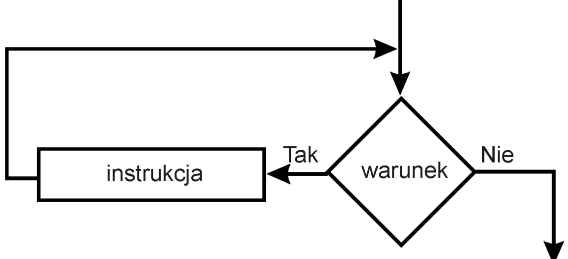
Wypisz 10
Wypisz 11
........
Wypisz 99

Algorytm ten składa się z 90 kroków i jest nieefektywny.

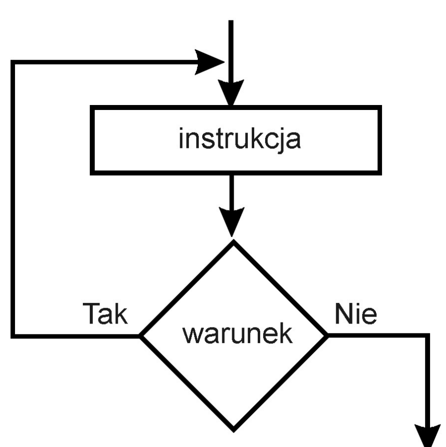
Zastosujmy tu **instrukcję iteracji (pętla)** – instrukcja powtarzania danego ciągu operacji. Liczba powtórzeń może być ustalona przed wykonaniem instrukcji lub może zależeć od spełnienia pewnego warunku, który jest sprawdzany w każdej iteracji.




Poniżej przedstawiamy trzy podstawowe przypadki iteracji stosowanych w algorytmach.



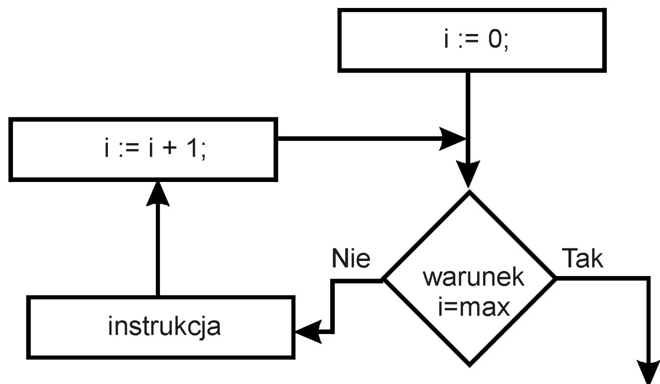

Najpierw sprawdzany jest warunek, potem wykonywana jest instrukcja. ( dopóki spełniony jest warunek wykonuj instrukcję - w Pascalu : While warunek do instrukcja )




Najpierw wykonywana jest instrukcja , a potem sprawdzany jest warunek ( wykonuj instrukcję dopóki spełniony jest warunek – w Pascalu : repeat instrukcja until wyrażenie ).

Instrukcja wykonywana jest z góry ustaloną (max) ilość razy ( w Pascalu : For zmienna := wyrażenie_1 to max do (downto) instrukcja )

W pętli tej wartość licznika „i” zwiększana jest o 1 po każdej wykonywanej instrukcji czyli jest **inkrementowana**

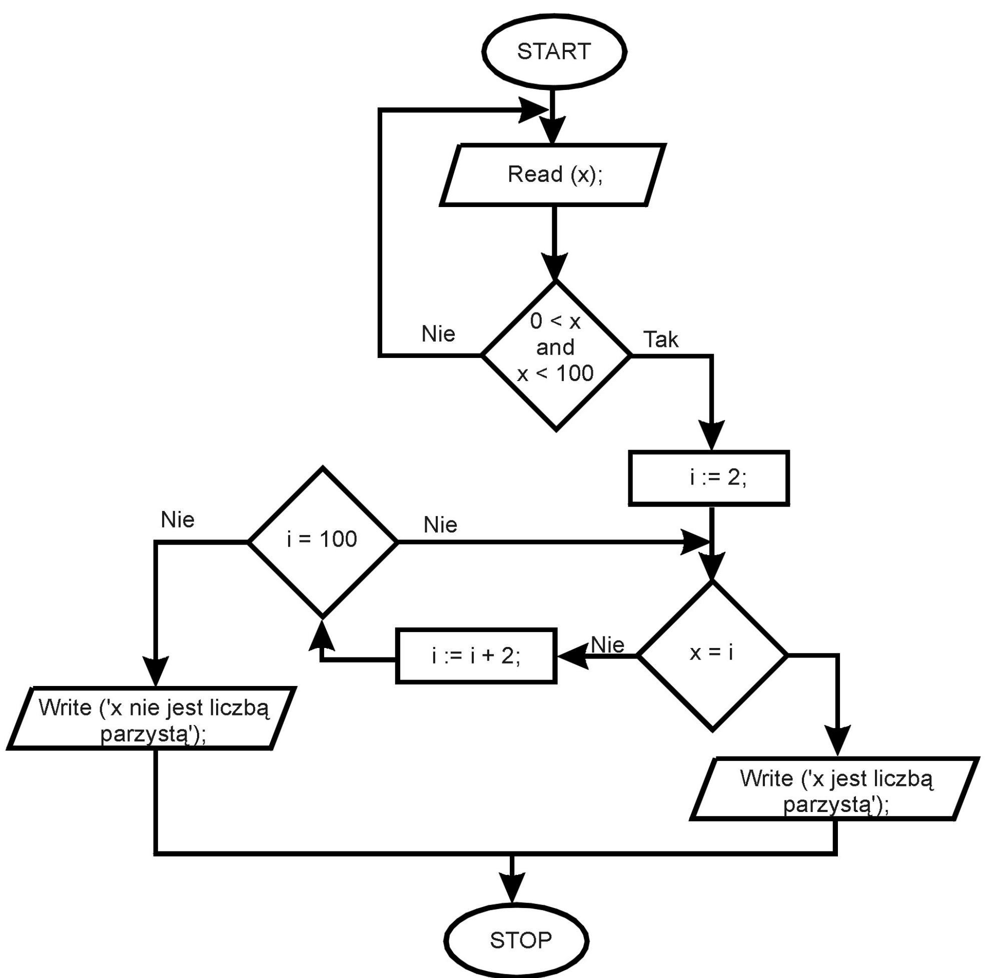


## *Złożoność obliczeniowa algorytmu.*

Podać algorytm, który pobiera na wejściu liczbę całkowitą dodatnią mniejszą od 100 i wyprowadza informację, czy jest to liczba parzysta czy nie.

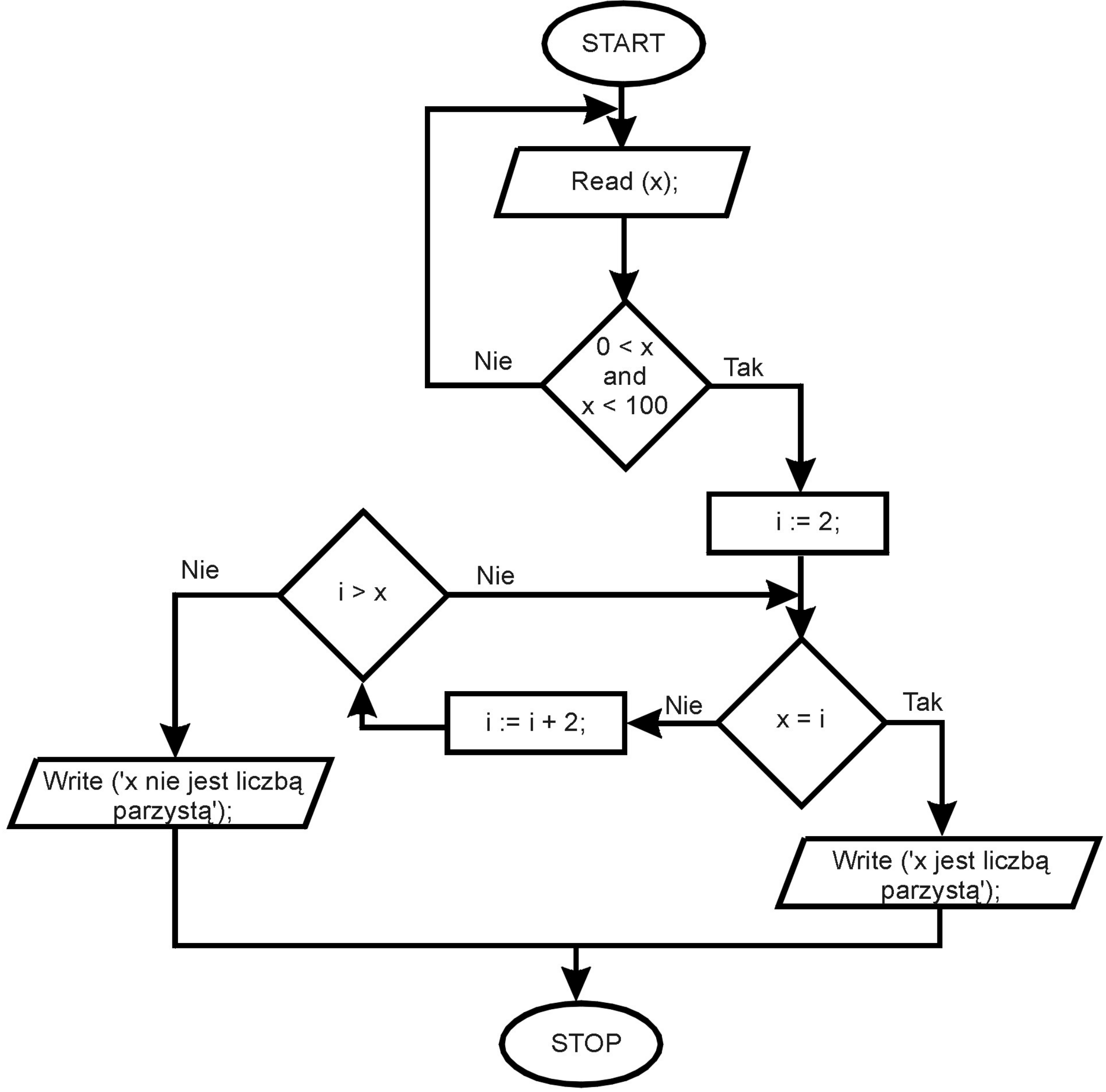
Problem algorytmiczny : badanie parzystości liczby podanej na wejściu
Dane wejściowe : $x \in N$ , $x < 100$ - badana liczba
Dane wyjściowe : napis „ x jest liczbą parzystą” gdy podana liczba jest parzysta lub napis „ x jest liczbą nieparzystą” gdy podana liczba jest nieparzysta
Zmienne pomocnicze : $i \in N$ - zmienna licznikowa

Algorytm 1




Opis : pobieramy liczbą i sprawdzamy warunek czy jest to liczba dodatnia i jednocześnie mniejsza od 100. Jeśli tak to porównujemy ją z każdą liczbą parzystą i począwszy od 2. Jeśli wejściowa liczba x jest liczbą nieparzystą to mamy w tym algorytmie maksymalnie 49 porównań tej liczby z liczbą „i”.

Algorytm 2




Podany algorytm 2 jest prawie identyczny z algorytmem 1 – jedyna różnica to porównywanie „i" nie z liczbą 100 ale z podaną liczbą x. W tym przypadku dla x nieparzystego > 1 liczba porównań będzie wynosiła (x-1)/2 lub dla x=1 jedno porównanie.

Czyli algorytm 2 jest algorytmem lepszym.

Oczywiście w przypadku podania liczby parzystej liczba porównań w obu algorytmach będzie jednakowa.

W celu oceny algorytmu wprowadzono pojęcie **złożoności obliczeniowej algorytmu** ( potocznie – algorytm, który rozwiązując to samo zadanie wykona mniej operacji).
Na złożoność obliczeniową algorytmu składa się :

- **złożoność pamięciowa –** zależy ona od wielkości pamięci komputera potrzebnej do realizacji algorytmu w postaci programu komputerowego. Złożoność pamięciowa jest najczęściej proporcjonalna do ilości zmiennych użytych w algorytmie.
- **złożoność czasowa –** pozwala oszacować czas potrzebny na wykonanie algorytmu.

Realizacja algorytmów w arkuszu kalkulacyjnym

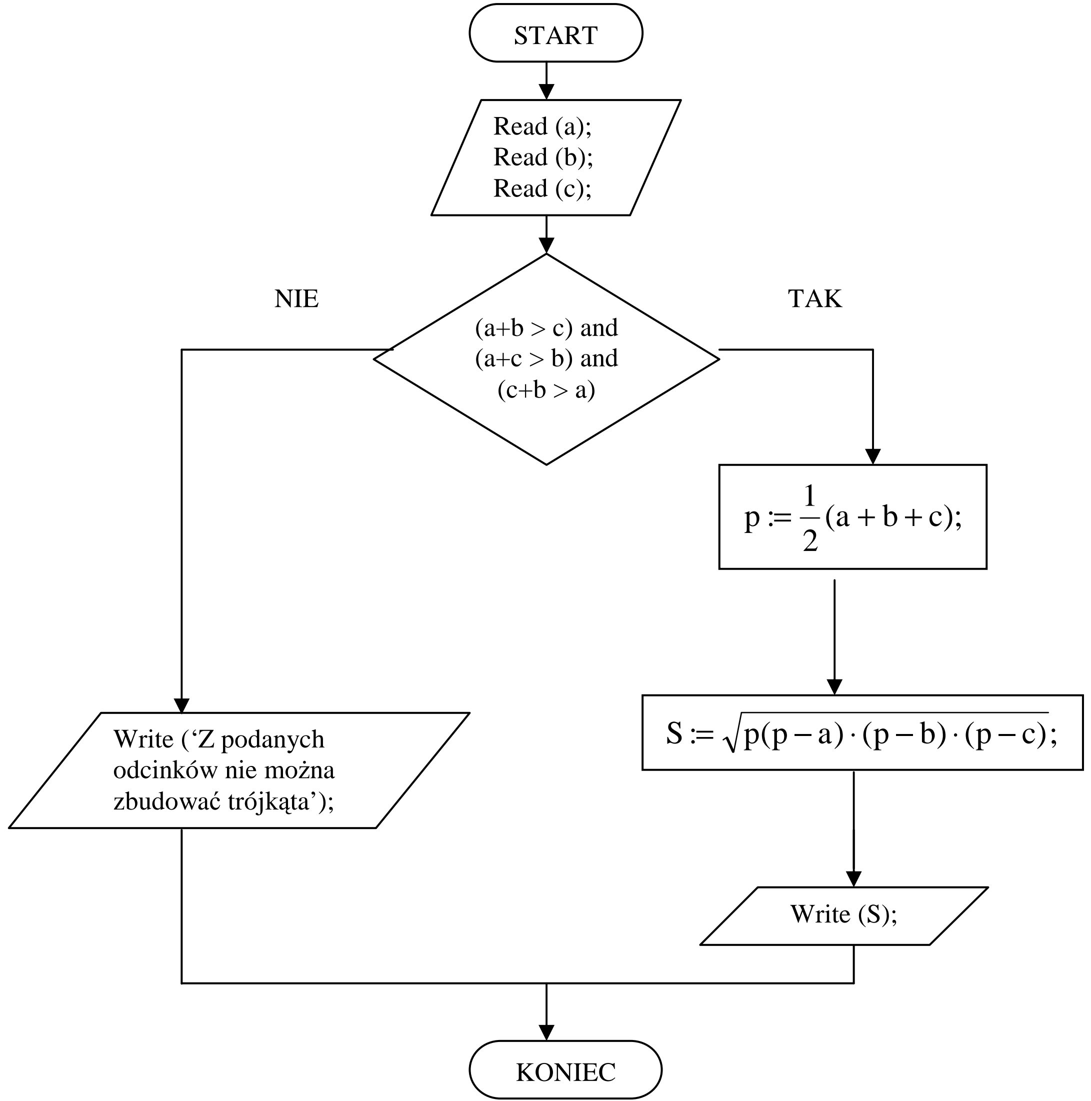
Problem algorytmiczny : Obliczenie pola powierzchni trójkąta na podstawie długości trzech odcinków ( zastosowanie wzoru Herona )

$$S = \sqrt{p(p-a)\cdot(p-b)\cdot(p-c)} \text{ gdzie } p = \frac{1}{2}(a+b+c).$$

Uwaga : trójkąt powstanie tylko wtedy, gdy długość boku największego jest mniejsza od sumy dwóch następnych boków

Dane wyjściowe : $S \in R^+$ – pole trójkąta o bokach a,b i c lub napis „ Z podanych odcinków nie można zbudować trójkąta”.

Zmienne pomocnicze : $p \in R^+$



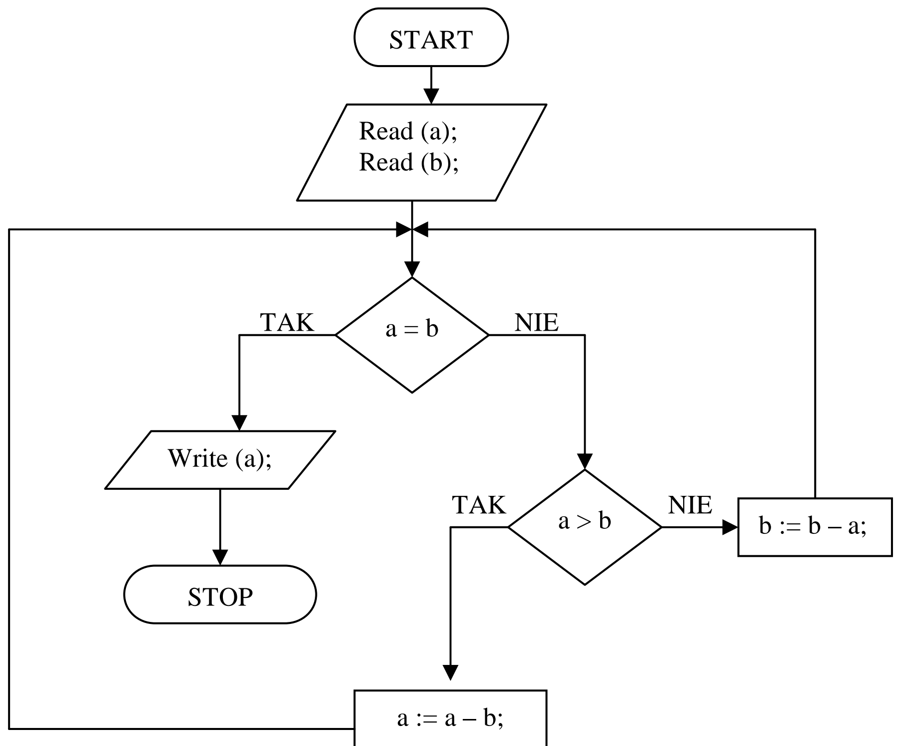

Jest to tak zwany **algorytm niestabilny**. W algorytmie takim błędy z zaokrągleń wyników pośrednich ( w tym wypadku „p” ) mogą wpływać na niedokładność wyników końcowych.

# 1. Wyznaczanie NWD i NWW – algorytm Euklidesa

a) Największy Wspólny Dzielnik

Pobieramy dwie liczby naturalne, od większej z nich odejmujemy mniejszą, a następnie większą liczbę zastępujemy różnicą. Postępujemy tak do momentu, gdy dwie liczby będą równe. Otrzymana liczba będzie NWD.
Przykład : a =12 , b = 20, ponieważ b > a to b = 20–12=8 , teraz a > b czyli a=12–8 =4, dalej b>a czyli b= 8–4 = 4 i a = 4 stąd NWD = 4

Problem algorytmiczny : największy wspólny dzielnik dwóch liczb
Dane wejściowe : a,b $\in$ N
Dane wyjściowe : NWD (a,b)

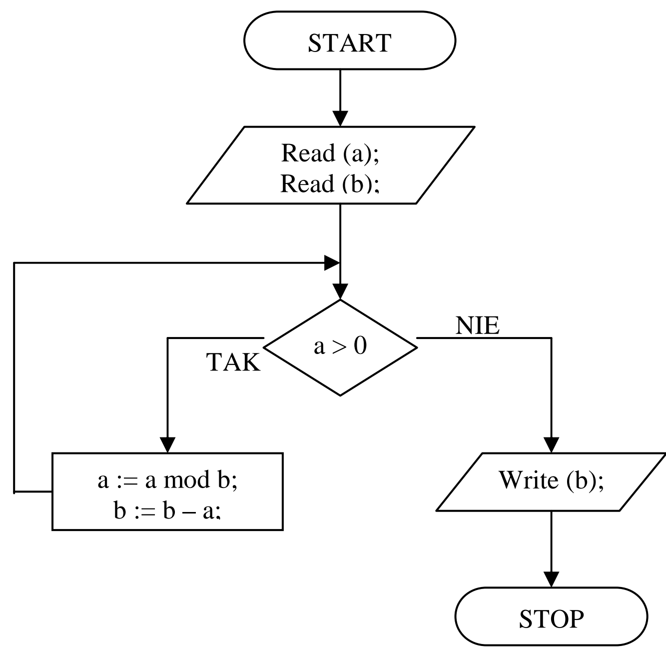



Zastosowano tu właściwości :
NWD (a,b) = NWD(a-b,b) jeśli a > b
NWD (a,b) = NWD(a,b-a) jeśli b > a

**Zmodyfikowany algorytm Euklidesa**

Równie często stosuje się modyfikację algorytmu NWD (a,b) = NWD(a mod b, b – (a mod b))

Będziemy iteracyjnie zmieniać wartości a i b aż do momentu, gdy a osiągnie wartość 0.
Przykład : a =12 , b = 20 , a = 12 mod 20 = 12 , b = 20 – 12 = 8 następnie a = 12 mod 8 = 4 , b = 8 – 4 = 4 , i następny krok a = 4 mod 4 = 0 czyli b = NWD = 4




b) Najmniejsza Wspólna Wielokrotność (dwóch liczb naturalnych)
(najmniejsza z liczb, która jest podzielna przez obie te liczby)

Algorytm ten opiera się na algorytmie NWD oraz na wzorze :
NWW (a,b) = (a*b) div NWD (a,b)

Ćwiczenie :
Posiadamy dwa czerpaki o pojemnościach 4 i 6 litrów, pusty pojemnik o dużej pojemności oraz nieograniczoną ilość wody (np. z kranu). Podaj sposób napełnienia pojemnika 15 litrami wody, przy czym woda może być wlewana lub wylewana z pojemnika tylko pełnymi czerpakami.

Rozwiązanie (ogólne) :

m – pojemność pierwszego czerpaka (tu 4 litry )
n – pojemność drugiego czerpaka ( tu 6 litrów )
k – pojemność do napełnienia ( tu 15 litrów )
x – liczba ruchów czerpaka 1
y – liczba ruchów czerpaka 2

Jeśli wlewamy wodę czerpakiem 1 to traktujemy to z plusem, jeśli wylewamy z naczynia to z minusem. Jeśli wlewaliśmy wodę 5 razy , a wylewaliśmy wodę 3 razy to liczba ruchów czerpaka wynosi 5-3 = 2. Jeśli natomiast wlewaliśmy 2 razy, a wylewaliśmy 5 razy to liczba ruchów czerpaka jest równa 2-5 = -3.

Tak więc aby problem był rozwiązany musi zostać spełnione równanie :
$mx + ny = k$

Pozostaje więc problem znalezienia dla ustalonych m i n (pojemności czerpaków) znalezienia odpowiednich wartości x i y (CAŁKOWITYCH).
Zwróćmy uwagę, że dla naszego ćwiczenia, równanie ma postać 4x+6y=15.
Już na pierwszy rzut oka widać, że równanie to nie ma rozwiązania w zbiorze liczb całkowitych ( Lewa strona r-nia jest parzysta, prawa nieparzysta).

A równanie 4x+8y = 10 ?
Równanie to także nie ma rozwiązania – lewa strona jest podzielna przez 4 , a prawa nie (pamiętajmy – liczby całkowite).

Stąd wniosek : równanie to ma rozwiązanie, gdy k=NWD (n,m). { ogólniej gdy z*k = NWD(n,m) }

No to zastosujmy algorytm Euklidesa NWD.

Zmodyfikujmy zadanie :

m = 12
n = 21
k = 3

12x + 21y = 9
Szukamy :
Wykorzystamy tu zależność : a = q*b + r    gdzie r  - reszta   0=< r < b
czyli   1)  21 = 1 * 12 + 9
2) 12 = 1 * 9 + 3
3)  9 = 3 * 3 + 0
stąd NWD (12,21) = 3

Z równania 2) wyznaczamy 3 (NWD)  12- 1*9 = 3
Z równania 1) wyznaczamy 9   i wstawiamy do równania 2)    21-1*12=9

12- 1*9 =3
12 – 1* (21-1*12) = 3
Czyli 2*12 – 1*21 = 3  Wlewamy 2 * 12 i wylewamy 21

**ZADANIE 1**

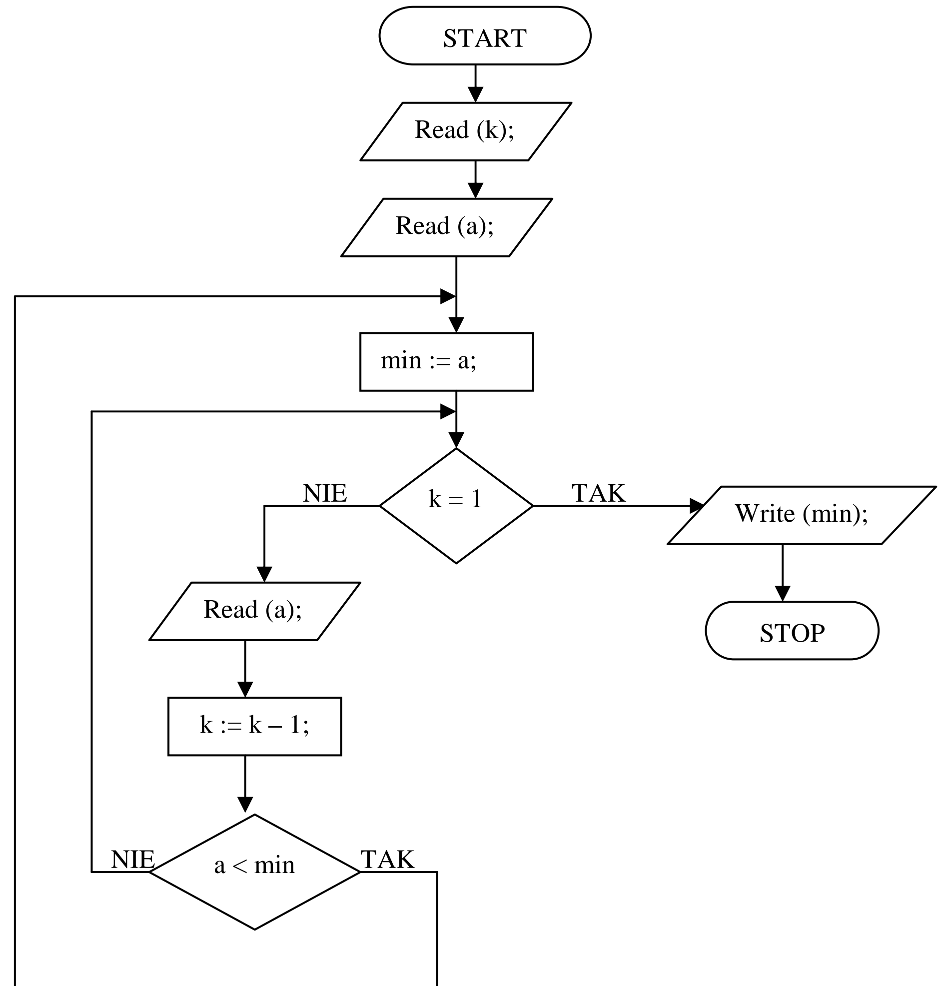
Za pomocą samodzielnie napisanego programu wygenerować plik tekstowy o nazwie „liczby.txt” zawierający 5001 liczb losowych z zakresu <0, 10 000>.

## 2. Największy i najmniejszy element zbioru.

**ZADANIE 2**

Znaleźć w pliku „liczby.txt” największą i najmniejszą liczbę.

Wykorzystamy tu , poznany wcześniej, zmodyfikowany schemat blokowy na znajdowanie najmniejszej (poniżej) i największej liczby.




W analogiczny sposób możemy znaleźć największy element w ciągu wprowadzanych kolejno liczb. Sprawdzamy tylko, czy kolejno wprowadzona liczba jest większa od aktualnie wprowadzonego maksimum.

Zwróćmy uwagę, że zastosowanie tych dwóch algorytmów nie jest jednoczesne, tzn. nie wyznaczamy największej i najmniejszej wielkości jednocześnie – przebiegamy dwukrotnie nasz zbiór. Ale możemy te dwa algorytmy połączyć i spowodować, że otrzymamy wartość maks i min jednocześnie. Zwróćmy uwagę, że w każdym z algorytmów wykonywane jest sprawdzenie czy x > y przy czym w algorytmie na maks x jest kandydatem , a w algorytmie na min y jest kandydatem.

***Algorytm w postaci listy kroków :***

Dane : zbiór n liczb
Wyniki : max i min , odpowiednio największy i najmniejszy element w zbiorze danych

Krok 1 : Podzielenie i połączenie elementów zbioru w pary. Jeśli n jest liczbą nieparzystą, to jeden z elementów pozostanie wolny – oznaczymy go liczbą z.
Krok 2 : Porównanie elementów w parze i dołączenie liczb do odpowiednich zbiorów. ( mamy parę liczb x i y. Przypuśćmy, że x > y , wobec tego x dołączamy do zbioru dla kandydatów na maks – zbiór M , a y do zbioru dla kandydatów na min – zbiór N )
Krok 3 : Znajdujemy max w zbiorze M za pomocą algorytmu na maksimum
Krok 4 : Znajdujemy min w zbiorze N za pomocą algorytmu na minimum
Krok 5 : Jeśli n jest liczbą nieparzystą to jeśli z < min to min = z , jeśli z > max to max = z.

Uwaga : Tak naprawdę wszystkie kroki realizujemy na jednym zbiorze danych przestawiając tylko elementy w parach tak aby element pierwszy w parze był większy od elementu drugiego w parze.
Przykład : Mamy zbiór liczb : 1,4,3,2,4,9,5,7

Krok 1 : podzielenie zbioru i połączenie elementów w pary : {1,4} , {3,2} , {4,9} , {5,7}
Krok 2 : porównanie elementów w parach : { 1 > 4 } { 3 > 2} { 4 > 9 } { 5 > 7 }

Jeśli prawda to pierwszy element z pary przenosimy do zbioru M , a drugi do N , jeśli fałsz to drugi element z pary przenosimy do M , a pierwszy do N.
Czyli nasz zbiór wygląda teraz tak : 4,1,3,2,9,4,7,5 – zwróć uwagę, że elementy o indeksie nieparzystym to zbiór M (maks) , a o indeksie parzystym to zbiór N (min).
Oczywiście można zamienić zbiory M i N ze sobą – nieparzyste to N , parzyste to M.

Krok 3 : znajdujemy w zbiorze M wartość max
Krok 4 : znajdujemy w zbiorze N wartość min
Krok 5 : opuszczamy - n liczbą parzystą.

Przedstawiony wyżej algorytm typu **dziel i zwyciężaj** jest najoptymalniejszym algorytmem wyszukiwania wartości maksymalnej i minimalnej w zbiorze.

## 3. Poszukiwanie lidera w zbiorze.

### ZADANIE 3

Znaleźć w pliku „liczby.txt” lidera zbioru.

Liderem nazywamy element, który występuje w zbiorze więcej niż połowę razy, czyli więcej niż n/2 razy, gdzie n jest liczbą elementów zbioru.
Problem znajdowania lidera w zbiorze ma na przykład zastosowanie w liczeniu głosów w wyborach, gdzie kandydat musi uzyskać więcej niż połowę głosów.

***Algorytm w postaci listy kroków :***
Algorytm ten składa się z dwóch etapów. W pierwszym szukamy kandydata na lidera, w drugim sprawdzamy, czy znaleziony kandydat jest rzeczywiście liderem.

Dane : zbiór $\{x_1, x_2 , ...... x_n\}$ złożony z n elementów
Wynik : lider L w zbiorze lub informacja, że zbiór nie zawiera lidera
Dane pomocnicze : c – krotność kandydata na lidera

Krok 1 : przyjmujemy pierwszy element zbioru za lidera oraz częstotliwość c = 1

{ etap 1 }

Krok 2 : dla kolejnych wartości i = 2,3,......, n wykonaj kroki 3-5 , a następnie przejdź do kroku 6

Krok 3 : jeśli c = 0 to wykonaj krok 4, a w przeciwnym wypadku krok 5

Krok 4 : {obieramy nowego kandydata na lidera } Przyjmujemy $x_i$ za lidera L , c=1

Krok 5 : {porównujemy kolejny element zbiór z kandydatem na lidera } Jeśli xi = L to c=c+1, w przeciwnym przypadku c = c – 1

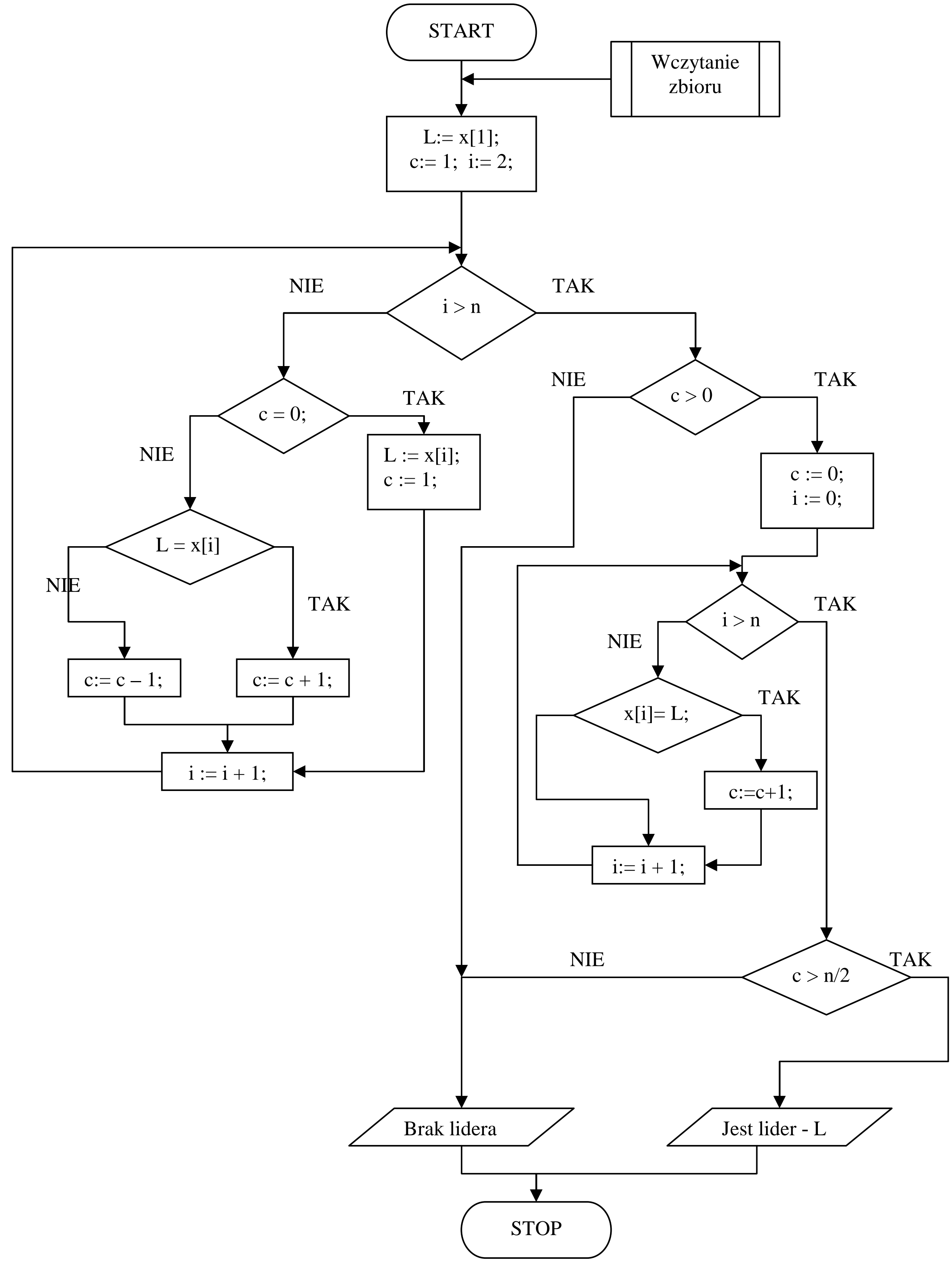
{etap 2 }

Krok 6 : jeśli c = 0 to przejdź do kroku 7, a w przeciwnym przypadku do kroku 8

Krok 7 : Zbiór nie ma lidera – koniec algorytmu

Krok 8 : wyznacz ile razy kandydat na lidera występuje w zbiorze. Jeśli liczba ta jest większa od n/2, to L jest rzeczywiście liderem, jeśli nie to zbiór nie ma lidera.

***Schemat blokowy***




## ZADANIE 5

Wyszukać w zbiorze „lista.txt" zadaną liczbę np. 232 oraz podać jej pozycję.

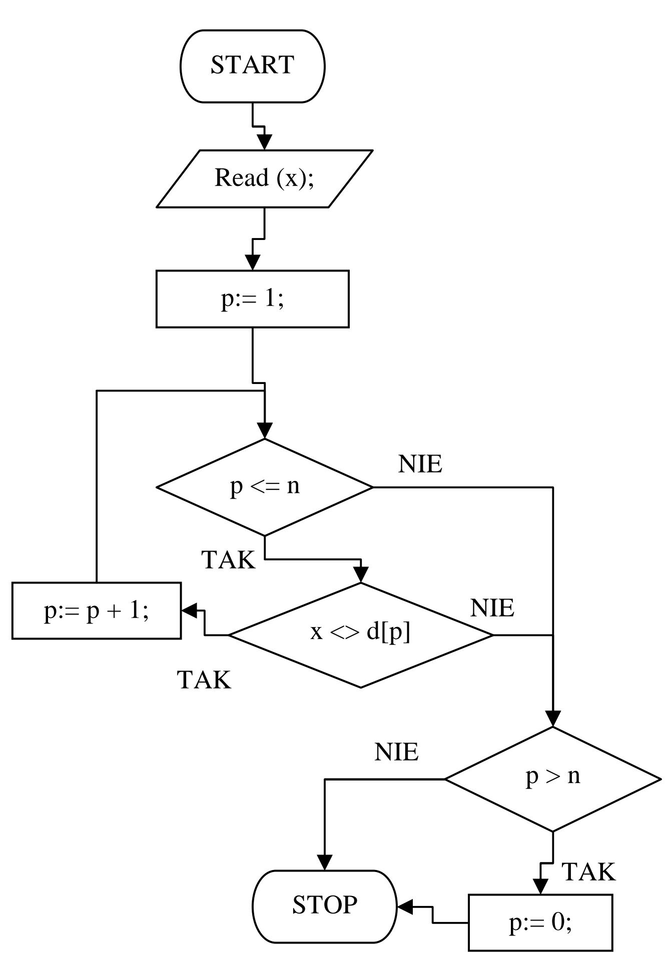
# 4. Przeszukiwanie sekwencyjne.

Zadanie przeszukiwania sekwencyjnego polega na przeglądaniu kolejnych elementów zbioru. Znaleziony element zostaje zwrócony (zwykle interesuje nas nie sam element, ale jego pozycja w zbiorze) lub algorytm zwraca informację, iż poszukiwanego elementu w zbiorze nie ma.
Ponieważ wymagane jest sprawdzenie kolejnych elementów zbioru, to w przypadku pesymistycznym (brak poszukiwanego elementu, lub jest on na samym końcu zbioru) algorytm musi wykonać n porównań, gdzie n jest liczbą elementów w zbiorze. Pesymistyczna złożoność czasowa jest następująca:

$$W(n) = n$$

***Specyfikacja algorytmu :***

*Dane wejściowe :*
n - liczba elementów w sortowanym zbiorze, $n \in N$
d[ ] - zbiór n-elementowy, który będzie przeszukiwany. Elementy zbioru mają indeksy od 1 do n.
x - wartość poszukiwana
*Dane wyjściowe :*
p - pozycja elementu *x* w zbiorze d[ ]. Jeśli *p* = 0, to element *x* w zbiorze nie występuje.




***Lista kroków***

Krok 1 : p:= 1;
Krok 2 : Dopóki (p ≤ n) and (x <> d[p]) wykonuj p := p + 1
Krok 3 : Jeśli p > n, to p :=0
Krok 4 : Zakończ algorytm

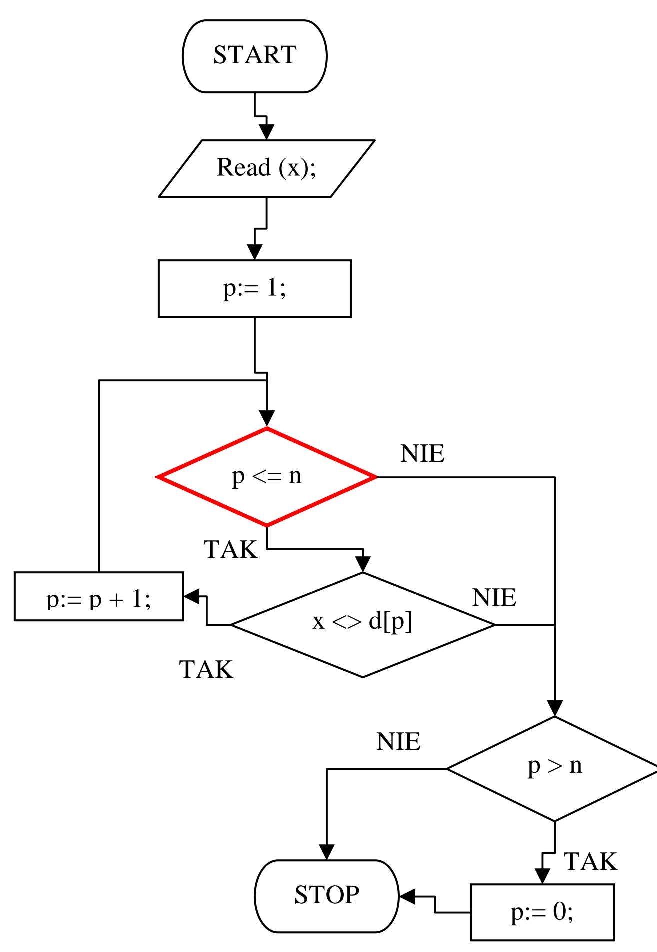
***Schemat blokowy.***
Przeszukiwanie zbioru rozpoczynamy od pierwszego elementu. Zmienna p przechowuje pozycję sprawdzanego elementu i na początku przyjmuje wartość 1.
W pętli warunkowej najpierw sprawdzamy, czy pozycja p wskazuje element wewnątrz zbioru. Jeśli nie, pętla jest przerywana.
Drugi test w pętli sprawdza, czy element zbioru leżący na pozycji p jest różny od poszukiwanego elementu x. Jeśli tak, to pozycja p jest zwiększana o 1 wskazując kolejny element zbioru i pętla kontynuuje się. Jeśli nie, to element został znaleziony i pętla zostaje przerwana.
Wyjście z pętli przeszukującej zbiór następuje w dwóch przypadkach:

1) przeglądnięty został cały zbiór, elementu x nie znaleziono i pozycja p jest większa od n.
2) znaleziono element x w zbiorze na pozycji p.

Zatem w przypadku gdy p > n, elementu x nie znaleziono i zerujemy pozycję p, aby zgłosić ten fakt. Inaczej p zawiera pozycję w zbiorze, na której znajduje się poszukiwany przez nas element x.

Ostatni test właściwie nie jest konieczny - program wykorzystujący ten algorytm może przecież sprawdzić, czy p jest większe od n. Jednakże wyzerowanie p jest bardziej czytelne i intuicyjne - w zbiorze nie ma elementu na pozycji zero. Zwracanie zera w przypadku niepowodzenia jest często stosowaną praktyką w programowaniu (jeśli zero jest możliwą wartością, to często zwraca się minus jeden dla zaznaczenia porażki)..

## 5. Wyszukiwanie z wartownikiem.




W algorytmie wyszukiwania sekwencyjnego znajduje się pętla wyszukująca. W pętli sprawdzane są dwa warunki. Pierwszy z nich (zaznaczony czerwoną linią na rysunku obok) jest potrzebny tylko wtedy, gdy zbiór d[ ] nie zawiera poszukiwanego elementu o wartości x. W takim przypadku wymieniony warunek gwarantuje zakończenie pętli przeszukującej po przeglądnięciu wszystkich elementów zbioru. Indeks p przyjmuje wartość n + 1.
Warunek jest sprawdzany dla każdego elementu zbioru. Jeśli moglibyśmy zagwarantować, iż poszukiwany element zawsze zostanie znaleziony, to wtedy warunek ten stałby się zbędny.
Możemy doprowadzić do takiej sytuacji dodając na końcu zbioru poszukiwany element. Wtedy pętla przeszukująca zakończy się albo na elemencie x leżącym wewnątrz zbioru, albo na elemencie x, który został dodany do zbioru. W pierwszym przypadku zmienna p będzie wskazywała pozycję znalezionego elementu i pozycja ta będzie mniejsza lub równa n, a w drugim przypadku (gdy element x w zbiorze nie występuje) zmienna p wskaże pozycję dodanego przez nas elementu, czyli n + 1. Ponieważ pętla ta zawsze zakończy się, to możemy pominąć pierwszy warunek pozostawiając jedynie test na różność od x. Uproszczenie to da w wyniku wzrost prędkości wyszukiwania.
Dodany przez nas element na końcu zbioru nosi nazwę wartownika (ang. guard lub sentinel), a stąd pochodzi nazwa algorytmu - przeszukiwanie z wartownikiem. Klasa złożoności obliczeniowej wynosi $\Theta(n)$.

### *Specyfikacja problemu*

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1. Zbiór musi posiadać miejsce na dodatkowy element, który zostanie dopisany na końcu.
x - wartość poszukiwana

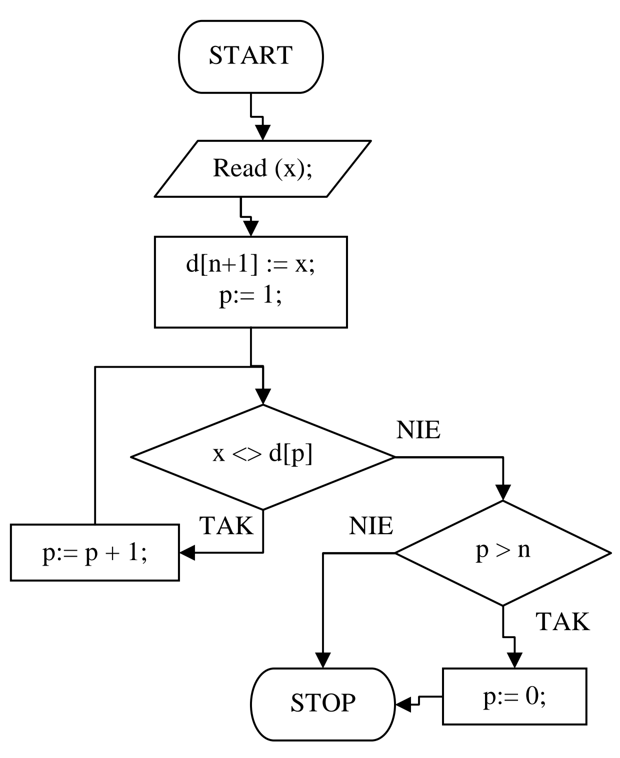
*Dane wyjściowe :*

p - pozycja elementu x w zbiorze d[ ]. Jeśli p = 0, to element x w zbiorze nie występuje

### *Lista kroków*

Krok 1 : d[n + 1] := x;
Krok 2 : p := 1;
Krok 3 : Dopóki x < > d[p]: wykonuj p := p + 1
Krok 4 : Jeśli p > n, to p := 0
Krok 5 : Zakończ algorytm

### *Schemat blokowy*




Na samym początku algorytmu dopisujemy do zbioru wartownika. Przeszukiwanie rozpoczynamy od pierwszego elementu, dlatego p przyjmuje wartość 1.
Pętla przeszukująca porównuje kolejne elementy zbioru z wartością poszukiwaną. Jeśli są różne, to przesuwa się do następnej pozycji. Wartownik gwarantuje nam, iż przeszukiwanie zawsze się zakończy.
Gdy pętla przeszukująca zostanie zakończona, zmienna p przechowuje pozycję elementu zbioru równego x. Jeśli p wskazuje wartownika, to zbiór nie zawiera poszukiwanego elementu - w takim przypadku zerujemy pozycję p, aby zgłosić porażkę. Kończymy algorytm.

***Przykładowy program :***

```
Program Szukaj_Z_Wartownikiem;
Uses Crt;
Const liczba_elementow = 5001;
Var pliczek         : Text;
    tablica_liczb   : Array [1..liczba_elementow+1] of Integer;
    liczba_txt      : String;
    p,liczba,code,x : Integer;

Begin
   ClrScr;
   Assign (pliczek,'liczby.txt');
   {$I-} { dyrektywa lokalna powodujaca, ze przy bledzie IO program sie nie "wysypie" }
   Reset (pliczek);
   {$I+}
   If IOResult = 0 Then
   Begin
      Writeln (' Wczytywanie danych do tablicy ');
      For p:= 1 to liczba_elementow do
      Begin
           Readln (pliczek,liczba_txt);
           Val (liczba_txt,liczba,code);
           If code = 0 Then tablica_liczb [p] := liczba;
      End;
      Close (pliczek);
      End Else Writeln ('Blad otwarcia pliku liczby.txt');

   Writeln;
   Write (' Podaj poszukiwana liczbe : ');
   Readln (x);
   tablica_liczb [liczba_elementow+1] := x;
   p:= 1;
   While x <> tablica_liczb [p] Do p:= p +1;
   If p > liczba_elementow Then Writeln (' Nie znaleziono liczby ',x,' w pliku')
      Else Writeln (' Poszukiwana liczba ',x,' zostala znaleziona na pozycji ',p);

   Repeat Until KeyPressed;
End.
```

**ZADANIE 6**

Znaleźć w pliku „liczby.txt" liczbę, która powtarza się najczęściej.

## 6. Wyszukiwanie najczęstszego elementu zbioru.

***Sposób 1.***

Pierwszym, narzucającym się rozwiązaniem jest podejście bezpośrednie. Wybieramy ze zbioru kolejne elementy i zliczamy ich wystąpienia. Wynikiem jest element, który występował najczęściej. Policzmy czasową złożoność obliczeniową takiego algorytmu: wybór n elementów wymaga n operacji. Po każdym wyborze musimy wykonać n porównań wybranego elementu z elementami zbioru w celu zliczenia ilości jego wystąpień. Daje to wynik: $T(n) = n^2$ . Zatem algorytm w tej postaci posiada klasę złożoności $O(n^2)$. W dalszej części zastanowimy się nad sposobami poprawy tego wyniku. (Zadanie o podobnej treści pojawiło się na maturze z informatyki w 2006 roku).
***Specyfikacja problemu***

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1. Zbiór musi posiadać miejsce na dodatkowy element, który zostanie dopisany na końcu.

*Dane wyjściowe :*

w_n - wartość elementu powtarzającego się najczęściej
p_n - pierwsza pozycja elementu najczęstszego
L_n - liczba wystąpień najczęstszego elementu

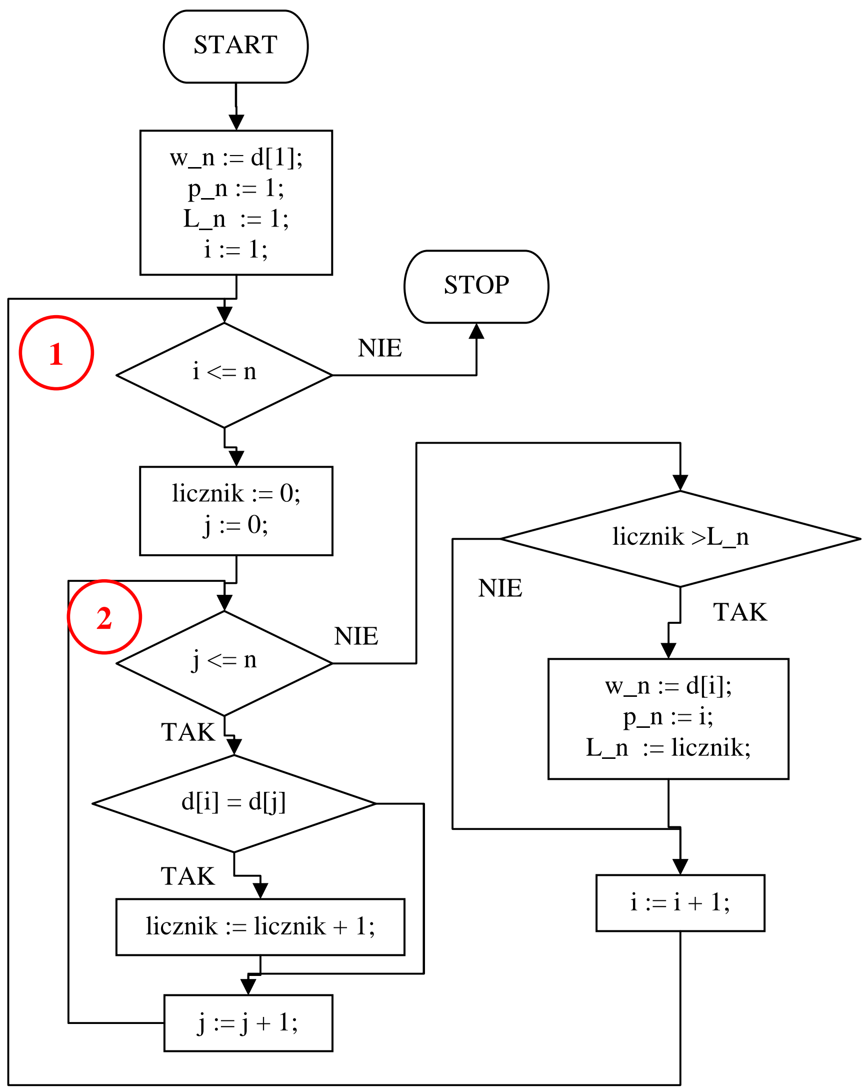
*Zmienne pomocnicze :*

i,j - zmienne licznikowe pętli
licznik - licznik wystąpień elementu

***Lista kroków***

Krok 1 : w_n := d[1]; p_n :=1; L_n := 1
Krok 2 : Dla i = 1,2,...,n: wykonuj kroki 3...5
Krok 3 : licznik := 0;
Krok 4 : Dla j = 1,2,...,n : jeśli d[i] = d[j], to licznik := licznik + 1;
Krok 5 : Jeśli licznik > L_n , to w_n := d[i]; p_n := i; L_n := licznik
Krok 6 : Zakończ algorytm

***Schemat blokowy***




Algorytm wyszukiwania najczęściej pojawiającego się elementu w zbiorze rozpoczynamy od inicjalizacji zmiennych, których zawartość będzie wynikiem pracy algorytmu.
Do w_n trafia pierwszy element zbioru - tymczasowo będzie to najczęstszy element. W p_n umieszczamy pozycję tego elementu, czyli 1. Do L_n zapisujemy liczbę wystąpień, też 1.
Rozpoczynamy pętlę nr 1, która wybiera ze zbioru kolejne elementy. W zmiennej licznik będziemy zliczać ilość wystąpień elementu d[i]. Dokonuje tego wewnętrzna pętla nr 2, która przegląda kolejne elementy zbioru i jeśli są równe elementowi d[i], zwiększa o 1 stan licznika. Po zakończeniu tej pętli w zmiennej licznik mamy liczbę wystąpień w zbiorze elementu d[i]. Jeśli licznik zawiera wartość większą od ilości powtórzeń tymczasowego elementu L_n, to za nowy tymczasowy element przyjmujemy d[i]. Do w_n trafia wartość elementu, do p_n zapisujemy numer jego pozycji, czyli i, a do L_n zapisujemy wyznaczoną w zmiennej licznik liczbę powtórzeń.
Na końcu pętli nr 1 zwiększamy i o 1, czyli przechodzimy do kolejnego elementu w zbiorze i operację zliczania powtarzamy.
Po zakończeniu pętli nr 1 w zmiennej w_n mamy wartość najczęściej powtarzającego się elementu, w p_n jest pozycja jego pierwszego pojawienia się w zbiorze, a w L_n mamy wyznaczoną ilość wystąpień.

Algorytm jest bardzo prosty w działaniu, lecz mało efektywny - niektóre elementy są zliczane wielokrotnie, a niektóre zupełnie niepotrzebnie. Usuwając te wady usprawnimy algorytm, co jest celem następnego rozdziału. W programowaniu często obowiązuje zasada:
*Efektywność algorytmu jest odwrotnie proporcjonalna do jego złożoności. Wynika stąd, iż proste algorytmy mogą być mało efektywne, podczas gdy algorytmy złożone działają bardzo szybko pomijając operacje zbędne.*

**Sposób 2.**

Główna pętla nr 1 w poprzednio opisanym algorytmie wybiera ze zbioru kolejne elementy. Następnie wewnętrzna pętla nr 2 zlicza ich wystąpienia poczynając od początku zbioru. Jeśli dokładnie rozważymy sytuację, to dojdziemy do wniosku, iż nie jest to konieczne. Otóż jeśli wybierzemy element na pozycji i-tej, a występował on już wcześniej w zbiorze, to algorytm zliczył jego wystąpienia. Nie ma sensu powtarzanie jeszcze raz tej operacji. Z drugiej strony jeśli element na pozycji i-tej jest nowym elementem, to wcześniej nie występował. W obu przypadkach wystarczy, jeśli zliczymy wystąpienia elementu poczynając nie od pozycji nr 1, lecz od (i + 1) z licznikiem równym 1:

1. dla elementu, który już wcześniej występował, otrzymamy mniejszą ilość wystąpień, lecz nie ma to żadnego znaczenia, ponieważ ten element został już przez algorytm przetworzony
2. dla nowego elementu otrzymamy poprawną ilość wystąpień

Ta drobna modyfikacja zredukuje ilość niezbędnych operacji porównań.

Następną optymalizacją jest ograniczenie zakresu pętli nr 1. Po pierwsze nie musimy sprawdzać ostatniego elementu, ponieważ nawet jeśli jest on elementem nowym, to występuje w zbiorze co najwyżej raz. Zatem możemy śmiało ograniczyć zakres pierwszej pętli do elementów od 1 do n - 1. Jeśli pójdziemy śmiało za ciosem, to okaże się, iż ilość wykonań pętli nr 1 można dalej ograniczyć. Załóżmy, iż w trakcie pracy algorytm wyznaczył pewien element zbioru, który powtarza się w nim L_n razy. W takim przypadku wystarczy zakończyć przeglądanie zbioru na pozycji n – L_n, ponieważ jeśli na końcowych L_n pozycjach występuje nowy element, to i tak nie będzie on częstszy od wyznaczonego wcześniej.
Podsumujmy nasze modyfikacje:

- Pętla wewnętrzna nr 2 przebiega pozycje od (i + 1) do n-tej
- Pętla zewnętrzna nr 1 przebiega elementy na pozycjach od 1 do n – L_n, gdzie L_n jest aktualną ilością wystąpień elementu najczęstszego

Drugie usprawnienie jeszcze bardziej zmniejsza ilość niezbędnych porównań.

***Specyfikacja problemu***

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1. Zbiór musi posiadać miejsce na dodatkowy element, który zostanie dopisany na końcu.

*Dane wyjściowe :*

w_n - wartość elementu powtarzającego się najczęściej
p_n - pierwsza pozycja elementu najczęstszego
L_n - liczba wystąpień najczęstszego elementu

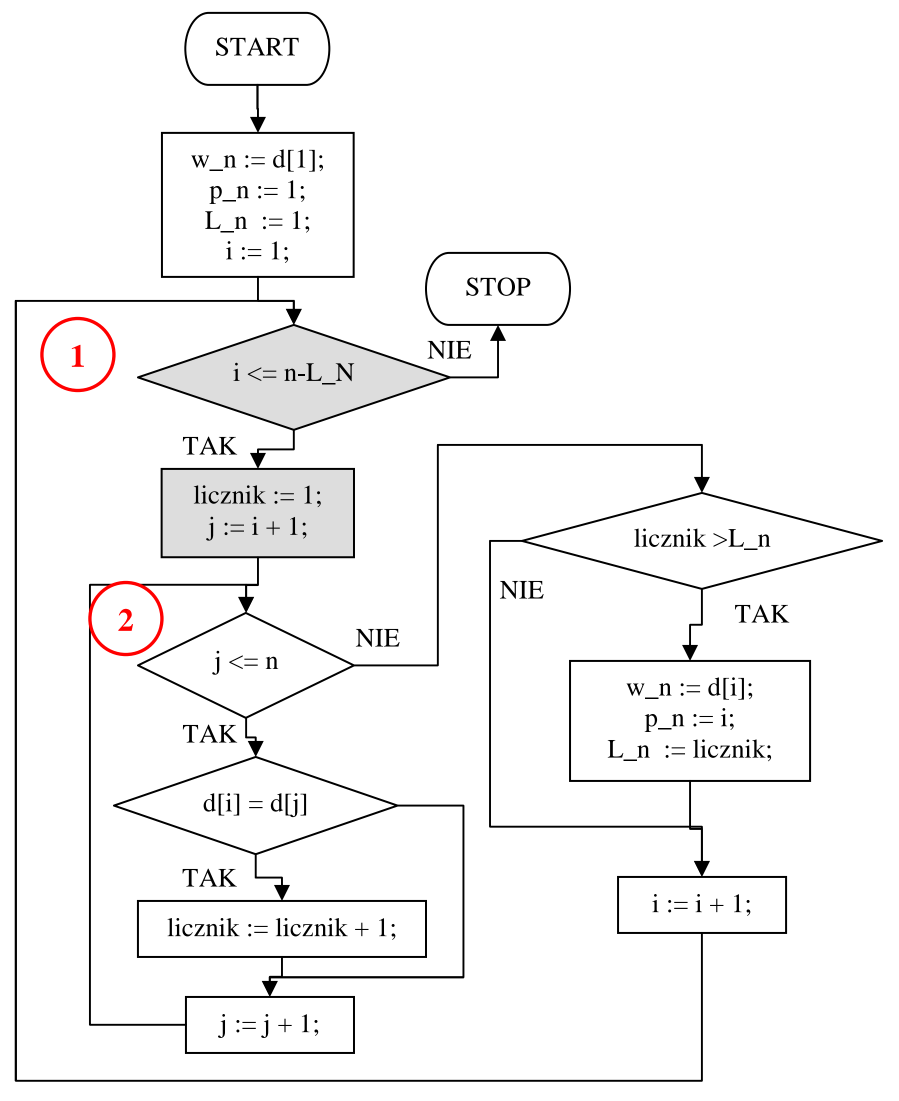
*Zmienne pomocnicze :*

i,j - zmienne licznikowe pętli
licznik - licznik wystąpień elementu

***Lista kroków***

Krok 1 : w_n := d[1]; p_n :=1; L_n := 1
Krok 2 : Dla i = 1,2,...,n – L_n : wykonuj kroki 3...5
Krok 3 : licznik := 1;
Krok 4 : Dla j = i+1, i+2,...,n : jeśli d[i] = d[j], to licznik := licznik + 1;
Krok 5 : Jeśli licznik > L_n , to w_n := d[i]; p_n := i; L_n := licznik
Krok 6 : Zakończ algorytm

***Schemat blokowy***




Przedstawiony obok algorytm wyszukiwania najczęstszego elementu zbioru jest jedynie drobną modyfikacją algorytmu z poprzedniej wersji - zmiany zaznaczono na schemacie blokowym innym kolorem tła symboli. Usprawnienia polegają na ograniczeniu liczby wykonań obu pętli.

Pętla nr 1 wyznacza elementy zbioru, których liczność jest wyliczana. Jej wykonanie ograniczamy do elementów na pozycjach od 1 do n - ln. Ostatnie ln elementów zbioru możemy pominąć, ponieważ nie wpływają one na wyznaczenie elementu najczęstszego.

Pętla nr 2 wylicza wystąpienia wybranych przez pętlę nr 1 elementów. Zliczanie rozpoczynamy od pozycji i + 1 do n. Istotne są jedynie nowe elementy, a elementy leżące na pozycjach mniejszych od i-tej zostały już przez algorytm zliczone. Pozostała część algorytmu jest niezmieniona.

**Sposób 3**

Jeśli elementy zbioru są liczbami całkowitymi (lub dadzą się sprowadzić do liczb całkowitych), to problem wyszukania najczęstszego elementu można rozwiązać w czasie liniowym wykorzystując dodatkowe struktury w pamięci. Zasada jest następująca: *dla każdej liczby ze zbioru przygotowujemy licznik częstości. Liczniki zerujemy. Przeglądamy zbiór zliczając w licznikach kolejno napotkane elementy. Następnie sprawdzamy, który z liczników zawiera największą wartość. Element skojarzony z tym licznikiem jest najczęstszym elementem zbioru.*

Podany algorytm jest szczególnie korzystny, gdy mamy wyznaczyć najczęstszy element wśród dużej liczby elementów (np. miliony lub dziesiątki milionów głosów) o niewielkim zakresie wartości (np. numery kandydatów w wyborach do sejmu lub senatu).

***Specyfikacja problemu***

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1. Zbiór musi posiadać miejsce na dodatkowy element, który zostanie dopisany na końcu.

*Dane wyjściowe :*

w_n - wartość elementu powtarzającego się najczęściej
L_n - liczba wystąpień najczęstszego elementu
w_t - true, jeśli istnieje element najczęstszy; false w przypadku braku takiego elementu

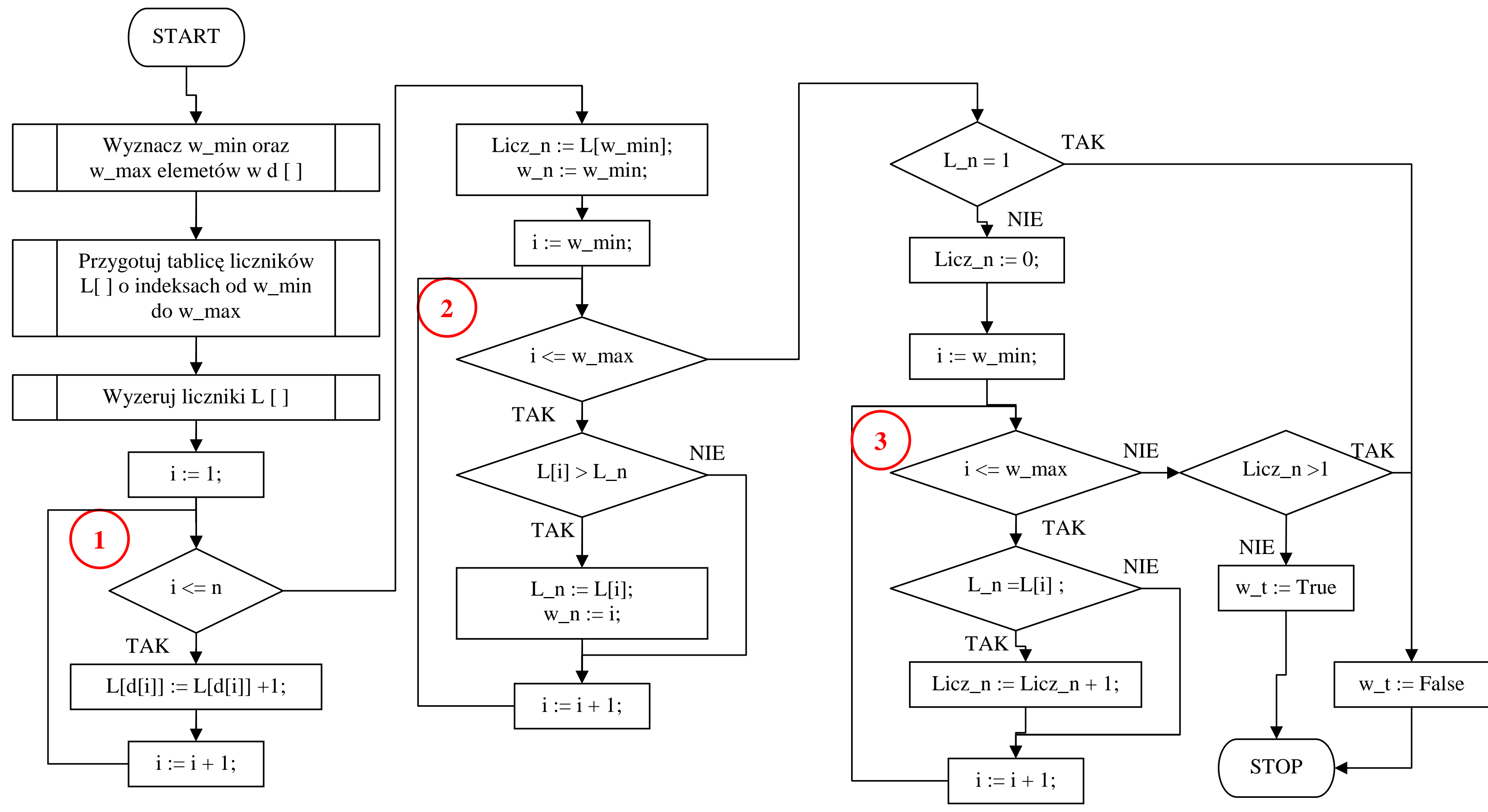
*Zmienne pomocnicze :*

i - zmienna licznikowa pętli
L[ ] - tablica liczników wystąpień elementów
Licz_n - zlicza ilość L_n w zbiorze liczników L[ ].

***Lista kroków***

Krok 1 : Wyznacz wartość minimalną w_min oraz wartość maksymalną w_max elementów w zbiorze d[ ].
Krok 2 : Przygotuj tablicę liczników L[ ] o indeksach od w_min do w_max
Krok 3 : Wyzeruj wszystkie liczniki w tablicy L[ ].
Krok 4 : Dla i = 1,2,...,n : L[d[i]] := L[d[i]] + 1
Krok 5 : L_n := L[w_min]; w_n := w_min;
Krok 6 : Dla i: = w_min, w_min + 1,...,w_max :
jeśli L[i] > L_n , to L_n := d[i]; w_n := i;
Krok 7 : Jeśli L_n = 1, to w_t := False i idź do kroku 12
Krok 8 : Licz_n := 0;
Krok 9 : Dla i = w_min, w_min + 1,...,w_max: jeśli L_n = L[i], to Licz_n := Licz_n + 1
Krok 10 : Jeśli Licz_n > 1, to w_t := False i idź do kroku 12
Krok 11 : w_t := True
Krok 12 : Zakończ algorytm

*Schemat blokowy*



W pierwszej kolejności wyznaczamy wartość minimalną wmin oraz wartość maksymalną wmax elementów zbioru d[ ]. Jeśli nie potrafimy oszacować tych wartości, to możemy zastosować opisany poprzednio algorytm jednoczesnego znajdowania wartości minimalnej i maksymalnej, który posiada liniową klasę złożoności obliczeniowej. Wartości te są nam potrzebne do przygotowania tablicy L[ ] zawierającej liczniki częstości, które następnie zerujemy. Liczniki będą zliczały występowanie każdej wartości całkowitej w przedziale od w_min do w_max.
Pętla nr 1 przebiega przez kolejne elementy zbioru d[ ] zliczając ich wystąpienia w odpowiednich licznikach. Po jej zakończeniu tablica L[ ] zawiera informację i ilości powtórzeń w zbiorze d[ ] każdej wartości z przedziału od w_min do w_max.

W kolejnym kroku wyszukujemy licznik, który zliczył największą ilość powtórzeń elementu - dokonuje tego pętla nr 2. W zmiennej L_n mamy maksymalną liczbę powtórzeń elementu o wartości w_n. Jeśli liczba ta wynosi 1, to żaden element zbioru nie powtarza się dwa razy, zatem ustawiamy wynik w_t na false i kończymy algorytm.
W przeciwnym razie sprawdzamy w pętli nr 3, czy w tablicy liczników L[ ] znajduje się licznik, który zliczył tyle samo powtórzeń. Może się tak zdarzyć, jeśli zbiór nie zawiera najczęstszego elementu, ale zawiera kilka elementów powtarzających się tyle samo razy. Jeśli znajdziemy taki licznik, to wyznaczona w zbiorze jest więcej niż jeden element powtarzający się L_n razy. Kończymy algorytm z wynikiem negatywnym. Jeśli natomiast tylko jeden licznik zliczył wartość L_n, to w zbiorze istnieje element najczęstszy i jest to w_n. W takim przypadku ustawiamy wynik w_t na true i kończymy algorytm.
W przeciwieństwie do algorytmów opisanych w dwóch poprzednich rozdziałach niniejszy algorytm nie tylko wyszukuje najczęstszy element, ale również sprawdza, czy znaleziony element jest faktycznie jedynym, najczęstszym elementem zbioru.

Klasa czasowej złożoności obliczeniowej jest równa $\Theta(n + m)$, gdzie n oznacza ilość elementów, a m jest ilością możliwych wartości elementów.
Klasa pamięciowej złożoności obliczeniowej jest równa $\Theta(m)$.

*Przykładowy program.*

```
program searchmf3;

const
  N    = 100;  { liczba elementow w zbiorze }
  W_MIN =  -9;  { dolny zakres elementow     }
  W_MAX =   9;  { gorny zakres elementow     }

var
  d              : array[1..N] of integer;
  L              : array[W_MIN..W_MAX] of integer;
  i,L_n,w_n,Licz_n : integer;
  w_t            : boolean;

begin
  writeln('Demonstracja wyszukiwania najczestszego elementu');
  writeln('------------------------------------------------');
  writeln('(C)2006 mgr Jerzy Walaszek       I LO w Tarnowie');
  writeln;

 { Generujemy zbior liczb pseudolosowych }

  randomize;
  for i := 1 to N do d[i] := W_MIN + random(W_MAX - W_MIN + 1);

 { Zerujemy liczniki }
```

```
  for i := W_MIN to W_MAX do L[i] := 0;

 { Zliczamy elementy zbioru }

  for i := 1 to N do inc(L[d[i]]);

 { Wyszukujemy licznik o najwiekszej liczbie powtorzen }

  L_n := L[W_MIN]; w_n := W_MIN;
  for i := W_MIN to W_MAX do
    if L[i] > L_n then
    begin
      L_n := L[i];
      w_n := i;
    end;

 { Sprawdzamy wyniki wyszukiwania }

  if L_n = 1 then w_t := false
  else
  begin
    Licz_n := 0;
    for i := W_MIN to W_MAX do if L[i] = L_n then inc(Licz_n);
    w_t := (Licz_n = 1);
  end;

{ Prezentujemy wyniki }

  for i := 1 to N do
    if w_t and (w_n = d[i]) then
      write('(',d[i]:2,')')
    else
      write(' ',d[i]:2,' ');
  writeln;
  if w_t then
    writeln('Element = ',w_n,', Liczba powtorzen = ',l_n)
  else
    writeln('W zbiorze brak elementu najczestszego');
  writeln;

{ Gotowe }

  writeln('Nacisnij klawisz Enter...');
  readln;
end.
```

**ZADANIE 7**

W pliku „liczby.txt” znaleźć drugą największą liczbę.

## *6.1. Drugi największy element zbioru.*

Jeśli zbiór jest uporządkowany, to zadanie staje się trywialnie proste - zwracamy przedostatni element. W przypadku zbioru nieuporządkowanego możemy zbiór posortować i zwrócić przedostatni element. Jednakże sortowanie jest kosztowne i zajmuje drogocenny czas. Istnieje o wiele prostsza metoda, która wymaga jednokrotnego przeglądnięcia zbioru, zatem ma klasę czasowej złożoności obliczeniowej $\Theta(n)$. Co więcej, nie zmienia ona struktury zbioru (kolejności elementów) oraz nie wymaga dodatkowej pamięci zależnej od ilości elementów w zbiorze.
Umówmy się, iż wynikiem poszukiwań będzie wartość elementu zbioru oraz jego położenie, czyli indeks. Dodatkowo nasz algorytm będzie zwracał wartość oraz położenie największego elementu w zbiorze.

***Specyfikacja problemu***

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1.

*Dane wyjściowe :*

w1 - wartość pierwszego największego elementu
w2 - wartość drugiego największego elementu
p1 - pozycja pierwszego największego elementu
p2 - pozycja drugiego największego elementu

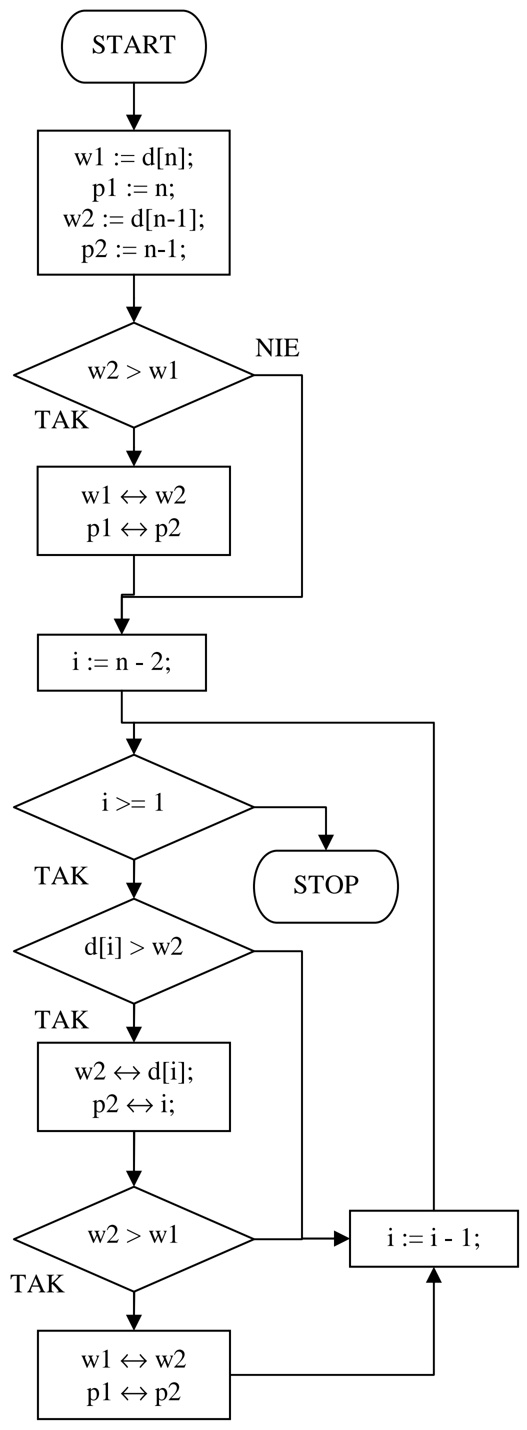
*Zmienne pomocnicze :*

i - zmienna licznikowa pętli

***Lista kroków***

Krok 1 : w1 := d[n]; p1 := n; w2 :=d[n - 1]; p2 := n - 1
Krok 2 : Jeśli w2 > w1 to w1 ↔w2; p1 ↔p2
Krok 3 : Dla i = n - 2, n - 3,...,1: wykonuj kroki 4...6
Krok 4 : Jeśli d[i] > w2, to idź do kroku 5 inaczej wykonaj następny obieg pętli z kroku 3
Krok 5 : w2 := d[i]; p2 := i;
Krok 6 : Jeśli w2 > w1, to w1 ↔ w2; p1 ↔p2
Krok 7 : Zakończ algorytm

*Schemat blokowy*




Na początku algorytmu wybieramy ze zbioru dwa ostatnie elementy i umieszczamy je odpowiednio w zmiennych w1 i w2. Dlaczego rozpoczynamy od końca zbioru? Otóż jeśli w przyszłości posortujemy zbiór stabilnym algorytmem sortującym (czyli takim, który nie zmienia kolejności elementów równych), to wyznaczony przez nasz algorytm drugi największy element będzie przedostatnim elementem zbioru posortowanego. Rozpoczynając przeglądanie od początku zbioru nie mielibyśmy takiej gwarancji (na przykład w przypadku, gdy w zbiorze są dwa elementy równe drugiemu największemu elementowi zbioru).

W zmiennych p1 i p2 zapamiętujemy pozycję tych elementów.

Umawiamy się, iż w1, p1 identyfikuje największy element w zbiorze, a w2 i p2 identyfikuje drugi największy element. Po inicjalizacji zmiennych musimy sprawdzić, czy wprowadzone do nich dane spełniają ten warunek. Jeśli nie, to wymieniamy ze sobą odpowiednie zawartości zmiennych.

Rozpoczynamy pętlę przeglądającą kolejne elementy zbioru od n - 2 do pierwszego. Pozycje n i n - 1 pomijamy, ponieważ odpowiadające im elementy są już w w1 i w2.

W każdym obiegu pętli sprawdzamy, czy i-ty element zbioru jest większy od tymczasowego drugiego największego elementu, czyli w2. Jeśli tak, to zapamiętujemy i-ty element w w2 oraz jego pozycję w p2. Teraz musimy sprawdzić, czy operacja ta nie spowodowała, iż w2 stało się większe od w1. Jeśli tak, to wymieniamy ze sobą zawartości zmiennych w1 z w2 oraz p1 z p2.

Po zakończeniu pętli w zmiennej w1 mamy wartość największego elementu zbioru, w p1 jego pozycję w zbiorze, w w2 mamy wartość drugiego największego elementu zbioru i w p2 jego pozycję. Algorytm kończymy.

## 6.2. K-ty największy element zbioru.

Jeśli k jest nieduże, to do wyszukania k-tego największego elementu możemy wykorzystać prosty algorytm będący rozwinięciem algorytmu z poprzedniego rozdziału. Otóż utworzymy listę, w której umieścimy k końcowych elementów zbioru. Lista będzie posortowana malejąco, tzn. element ostatni będzie najmniejszy, a pierwszy największy. Przeglądanie zbioru rozpoczniemy od elementu na (n - k)-tej pozycji do elementu na pozycji nr 1. Jeśli wybrany element zbioru będzie większy od ostatniego elementu listy, to dopiszemy go do tej listy tak, aby wciąż była uporządkowana (problem wstawiania elementu na listę uporządkowaną opisaliśmy w artykule o algorytmach sortujących - sortowanie przez wstawianie). Po operacji wstawienia ostatni element listy będzie zawsze usuwany - dzięki temu lista będzie zawierała zawsze dokładnie k elementów. Po przeglądnięciu całego zbioru na liście znajdzie się k kolejnych, największych elementów, przy czym element pierwszy będzie zawierał element największy, a element k-ty będzie zawierał k-ty największy element w zbiorze. Tego typu algorytm ma klasę złożoności $\Theta(k^2 + kn)$. Jeśli k jest stałe, a zmienia się n, to klasa złożoności redukuje się do $\Theta(n)$.

***Specyfikacja problemu***

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1.
k - numer poszukiwanego największego elementu, k < n

*Dane wyjściowe :*

naj [ ] - k elementowa lista zawierająca kolejne, największe elementy zbioru d[ ]. Indeksy rozpoczynają się od 1.
Element listy jest rekordem o dwóch polach:
w - pole zawiera wartość kolejnego elementu największego
p - pole zawiera pozycję kolejnego elementu największego
Elementy listy określają kolejne największe elementy w następujący sposób:
naj[1] - największy element
naj[2] - drugi największy element
...............
naj[k] – k-ty największy element

*Zmienne pomocnicze :*

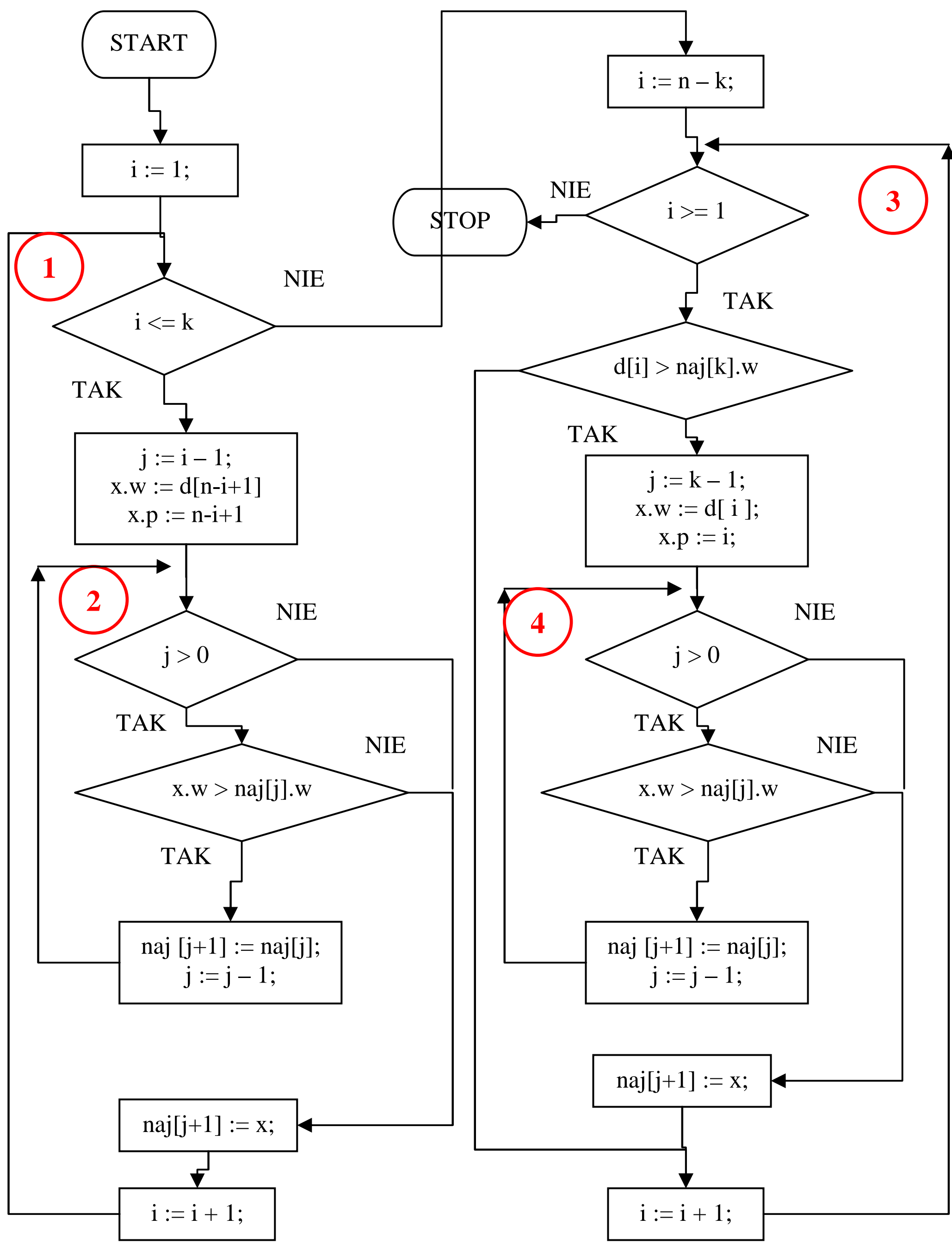
i,j - zmienne licznikowe pętli
x - zmienna pomocnicza do sortowania listy naj[ ]

***Lista kroków***

Krok 1 : Dla i = 1,2,...,k: wykonuj kroki 2...5
Krok 2 : j := i - 1; x•w := d[n - i + 1]; x•p := n - i + 1
Krok 3 : Dopóki (j > 0) and (x•w > naj[j]•w): wykonuj krok 4
Krok 4 : naj[j + 1] := naj[j]; j := j – 1;
Krok 5 : naj[j + 1] := x;

Krok 6 : Dla i = n - k, n - k - 1,...,1: wykonuj kroki 7..11
Krok 7 : Jeśli d[i] <= naj[k] •w , to wykonaj następny obieg pętli z kroku 6
Krok 8 : j := k - 1; x•w := d[i]; x•p := i;
Krok 9 : Dopóki (j > 0) and (x•w > naj[j]•w): wykonuj krok 10
Krok 10 : naj[j + 1] := naj[j]; j := j – 1;
Krok 11 : naj[j + 1] := x;
Krok 12 : Zakończ algorytm

***Schemat blokowy***




Algorytm wyznaczania k-tego największego elementu składa się z dwóch sortowań. Pierwsze sortowanie realizowane jest przez pętle nr 1 i pętlę nr 2. Celem jest utworzenie posortowanej listy

naj[ ] zawierającej k ostatnich elementów zbioru d[ ]. Lista jest sortowana malejąco, zatem pierwszy jej element będzie się odnosił do największego elementu, a ostatni do k-tego największego elementu.
Drugie sortowanie realizują pętle nr 3 i nr 4. W tym przypadku pętla nr 3 wybiera ze zbioru elementy poczynając od elementu o indeksie n - k i idąc w kierunku początku zbioru. Wybrany element jest porównywany z końcem listy naj[ ]. Jeśli jest większy od ostatniego elementu tej listy, to powinien się na niej znaleźć. Pętla nr 4 szuka miejsca wstawienia na listę naj[ ] wybranego elementu. Po jej zakończeniu wstawiamy element na wyznaczone miejsce na liście. Zwróć uwagę, iż lista naj[ ] nie rozrasta się. Zawsze ostatni element jest tracony i dlatego lista zachowuje długość k elementów.
Po zakończeniu pętli nr 3 na liście mamy zebrane k największych elementów w zbiorze (lista zapamiętuje zarówno wartość elementu jak i jego pozycję w zbiorze). k-ty największy element znajduje się na k-tej pozycji na tej liście. Algorytm kończy się.

Zaletą algorytmu jest to, iż zbiór wejściowy nie jest modyfikowany i zachowuje oryginalną kolejność elementów - czasem może to być pożądane. Dla małych wartości k algorytm ma dosyć dużą efektywność i nie wymaga wiele dodatkowej pamięci.

## 6.3. K-ty największy element zbioru – wyszukiwanie szybkie .

Istnieje szybki algorytm wyszukiwania k-tego największego elementu oparty na podziałach zbioru na partycje i posiadający pesymistyczną złożoność czasową $\Theta(n^2)$. W przypadku typowym k-ty największy element znajdowany jest w czasie liniowo logarytmicznym O(n log n). Algorytm nosi nazwę Szybkiego Wyszukiwania (ang. Quick Select) i został opracowany przez prof. Tony'ego Hoare'a (twórca znanego algorytmu sortującego Quick Sort).
Idea działania algorytmu szybkiego wyszukiwania jest genialnie prosta i opiera się na wykorzystaniu metody Dziel i Zwyciężaj (ang. Divide and Conquer):
W zbiorze wybieramy element środkowy tak zwany **pivot** leżący na pozycji w środku zbioru i dzielimy względem niego zbiór na dwie partycje tak, aby w lewej partycji znalazły się wszystkie elementy mniejsze od wybranego elementu, a w partycji prawej wszystkie elementy większe lub równe mu. Po podziale (opisanym dokładnie w artykule o algorytmach sortujących przy opisie algorytmu QuickSort) otrzymujemy wynikową pozycję elementu dzielącego. Jeśli jest równa k-tej pozycji od końca zbioru, to algorytm kończymy - element podziałowy jest poszukiwanym k-tym największym elementem w zbiorze. Jeśli równość nie zachodzi, to za nowy zbiór wybieramy tę partycję, która zawiera pozycję k-tą od końca zbioru i procedurę podziału powtarzamy aż do osiągnięcia pożądanego wyniku.
Poniżej przedstawiamy trzy możliwe przypadki, które mogą wystąpić po podziale zbioru na dwie partycje względem wybranego elementu:

| | | | | |
|---|---|---|---|---|
| 1. | lewa partycja | **j**<br>**k** | prawa partycja | Pozycja elementu podziałowego j-ta jest równa k-tej od końca pozycji w zbiorze. Zatem prawa partycja zawiera k - 1 elementów większych lub równych elementowi podziałowemu, który jest k-tym największym elementem |
| 2. | lewa partycja<br>**k** | **j** | prawa partycja | Pozycja j-ta jest większa od k-tej od końca pozycji w zbiorze. Poszukiwany element będzie zatem w lewej partycji. Za nowy zbiór przyjmiemy lewą partycję. |
| 3. | lewa partycja | **j** | prawa partycja<br>**k** | Pozycja j-ta jest mniejsza od k-tej od końca pozycji w zbiorze. Poszukiwany element będzie w prawej partycji. Za nowy zbiór przyjmiemy prawą partycję. |

***Specyfikacja problemu***

*Dane wejściowe :*

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1.
k - numer poszukiwanego największego elementu, k < n

*Dane wyjściowe :*

d [ ] - zbiór wejściowy tak uporządkowany, iż na *k*-tej od końca pozycji znajduje się poszukiwany element

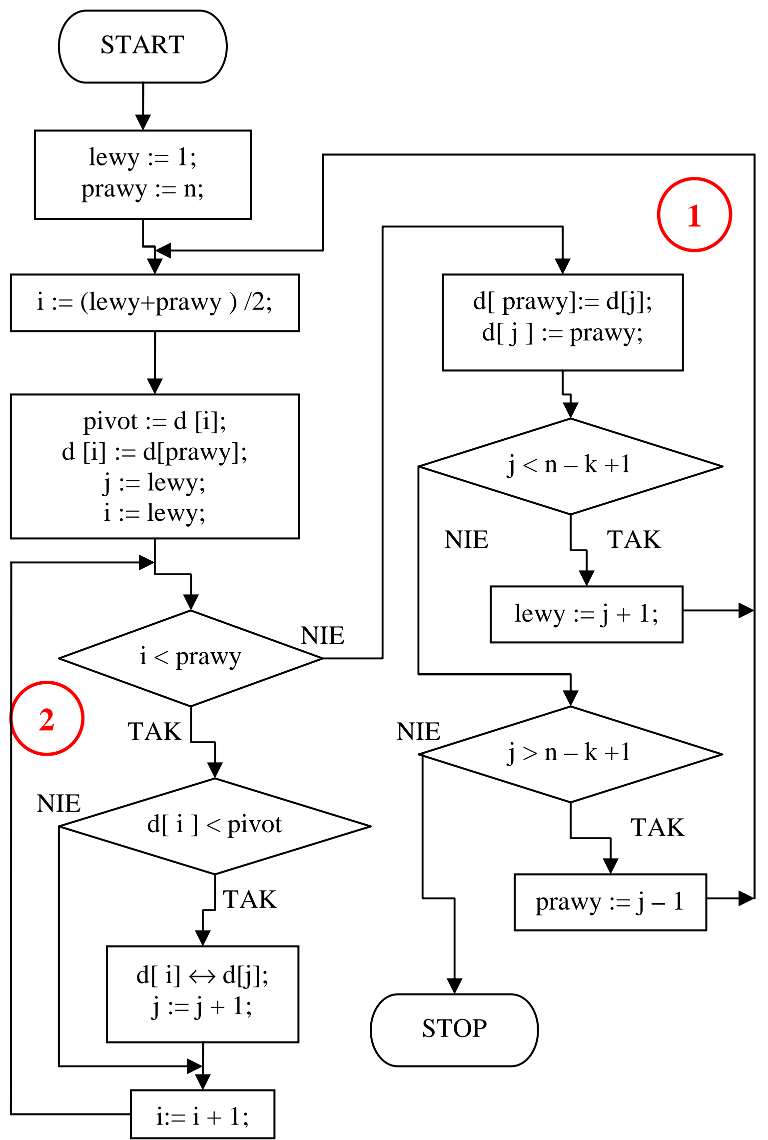
*Zmienne pomocnicze :*

i,j - wskaźniki wykorzystywane przy podziale zbioru na partycje; i, j ∈ N
lewy,
prawy - indeksy pierwszego i ostatniego elementu partycji. lewy, prawy ∈ N
pivot - element, wg którego zbiór będzie dzielony na partycje

***Lista kroków***

Krok 1 : lewy := 1; prawy := n;
Krok 2 : i := (lewy+prawy) / 2;
Krok 3 : pivot := d [i ]; d [i ] := d [prawy]; j := lewy
Krok 4 : Dla i:= lewy, lewy + 1, ..... prawy – 1 wykonuj kroki 5 i 6
Krok 5 : Jeśli d[i] >= pivot to wykonaj następny obieg pętli 4
Krok 6 : d [ i ] ↔ d [ j ]; j := j +1;
Krok 7 : d [ prawy ] := d [ j ]; d [ j ] := pivot;
Krok 8 : Jeśli j < n - k + 1, to lewy := j + 1 i wróć do kroku 2 , w przeciwnym razie wykonaj krok 9
Krok 9 : Jeśli j > n - k + 1, to prawy := j – 1 i wróć do kroku 2, w przeciwnym razie wykonaj krok 10
Krok 10 : Zakończ algorytm

*Schemat blokowy*




Na początku algorytmu ustawiamy granice dzielonej partycji tak, aby obejmowała cały zbiór.
Pętla nr 1 wyznacza k-ty największy element zbioru. Najpierw wyliczamy pozycję środowego elementu i zapamiętujemy ją w zmiennej i. Robimy to po to, aby uniknąć tych samych rachunków.
W zmiennej piwot umieszczamy element środkowy, a na jego miejsce przenosimy ostatni element. Dzięki tej operacji element środkowy zostaje usunięty ze zbioru (pamiętamy go w zmiennej piwot). Element ten posłuży do podziału zbioru na dwie partycje.
W zmiennej j zapamiętujemy pozycję podziałową. Na początku jest to pozycja pierwszego elementu zbioru.
W pętli nr 2 przeglądamy kolejno elementy zbioru od pierwszego do przedostatniego (ostatni zapisaliśmy na pozycji elementu środkowego). Jeśli element d[i] jest mniejszy od piwotu, to zamieniamy go z elementem na pozycji podziałowej. Po operacji pozycja podziałowa zostaje przesunięta o jeden element. Zwróć uwagę, iż w wyniku wszystkie elementy poprzedzające d[j] są mniejsze od wybranego na początku elementu środkowego.
Po zakończeniu pętli nr 2 zbiór jest podzielony na dwie partycje. Pozostaje nam jedynie zwolnienie pozycji j-tej na element podziałowy piwot. Zatem element z pozycji j-tej przenosimy na koniec zbioru, a na pozycji j-tej umieszczamy element podziałowy.
Lewa partycja rozciąga się od pozycji lewy do j - 1. Zawiera ona elementy mniejsze od elementu podziałowego.
Prawa partycja rozciąga się od pozycji j + 1 do prawy i zawiera elementy większe lub równe elementowi podziałowemu.
Sprawdzamy, czy pozycja k-ta od końca występuje w partycji lewej lub w prawej. Jeśli tak, to za nowy zbiór przyjmujemy odpowiednią partycję i kontynuujemy jej podział za pomocą pętli nr 1. Jeśli natomiast pozycja k-ta od końca zbioru nie wpada w żadną z wyznaczonych partycji, to musi być równa wyznaczonej pozycji elementu podziałowego, czyli j. Skoro tak, to element podziałowy jest poszukiwanym k-tym największym elementem w zbiorze - kończymy algorytm. Wartość elementu możemy odczytać z k-tej od końca pozycji.

*Przykładowy program.*

```
Program Quick_K_ty_Element;
Uses Crt;
Const liczba_elementow = 5001;

Var
   tablica_liczb : Array [1..liczba_elementow] of Integer;
   k             : Integer;
{----------------------------------------------------------------------}

Function Wczytaj_Liczby : Boolean;

Var pliczek         : Text;
   i,liczba,code   : Integer;
   liczba_txt      : String;

Begin
  Wczytaj_Liczby := False;
  Assign (pliczek,'liczby.txt');
  {$I-} { dyrektywa lokalna powodujaca, ze przy bledzie IO program sie nie "wysypie" }
  Reset (pliczek);
  {$I+}
  If IOResult = 0 Then
  Begin
    Writeln (' Wczytywanie danych do tablicy ');
    For i:= 1 to liczba_elementow do
    Begin
        Readln (pliczek,liczba_txt);
        Val (liczba_txt,liczba,code);
        If code = 0 Then tablica_liczb [i] := liczba;
    End;
    Close (pliczek);
    Wczytaj_Liczby := True;
  End Else Writeln ('Blad otwarcia pliku liczby.txt');
End;
{----------------------------------------------------------------------}
Procedure Szukaj_K;
Var i,j,lewy,prawy,pivot,x : Integer;
   koniec                  : Boolean;
Begin
     lewy  := 1;
     prawy := liczba_elementow;
     koniec := False;
     While koniec = False Do
     Begin
         i := (lewy + prawy) div 2;
         pivot := tablica_liczb [i];
         tablica_liczb [i]  := tablica_liczb [prawy];
         j := lewy;
         For i := lewy To prawy - 1 Do
          If tablica_liczb [i] < pivot Then
          Begin
              x := tablica_liczb [i];
              tablica_liczb [i] := tablica_liczb [j];
              tablica_liczb [j] := x;
              Inc(j);
          End;
         tablica_liczb [prawy] := tablica_liczb [j];
         tablica_liczb [j] := pivot;
         If j < liczba_elementow - k + 1 Then lewy := j + 1
```

```
        Else if j > liczba_elementow - k + 1 Then prawy := j - 1
         Else koniec := True;
    End;
End;

{----------------------------------------------------------------------}

Procedure Zapisz_Do_Pliku;
Var pliczek       : Text;
   i              : Integer;
Begin
    Assign (pliczek,'K_Quick.txt');
    Rewrite (pliczek);
    For i:= 1 to liczba_elementow do Writeln (pliczek,tablica_liczb [i]);
    Close (pliczek);
End;
{----------------------------------------------------------------------}

Begin
    ClrScr;
    Writeln;
    Writeln('Szybkie wyszukiwanie k-tego najwiekszego elementu');
    Writeln;
    Write (' Podaj k (od 1 do 9) ');
    Readln (k);
    Writeln;
    If Wczytaj_Liczby Then
    Begin
        Szukaj_K;
        Writeln;
        Writeln(k,' najwiekszy element = ',tablica_liczb[liczba_elementow - k + 1]);
        Writeln;
        Writeln (' Zapis tablicy do pliku K_Quick.txt ');
        Zapisz_Do_Pliku;
        Writeln;
        Writeln (' Koniec - Nacisnij dowolny klawisz');
        Repeat Until KeyPressed;
    End;
End.
```

W programie tym dodatkowo zrealizowano zapis tablicy do pliku – można zauważyć, że program ten zmienia kolejność elementów w zbiorze.

# 7. Sito Erastotenesa.

## ZADANIE 8

Wyszukać w zbiorze „lista.txt" wszystkie liczby pierwsze i zapisać je do zbioru tekstowego „pierwsze.txt".

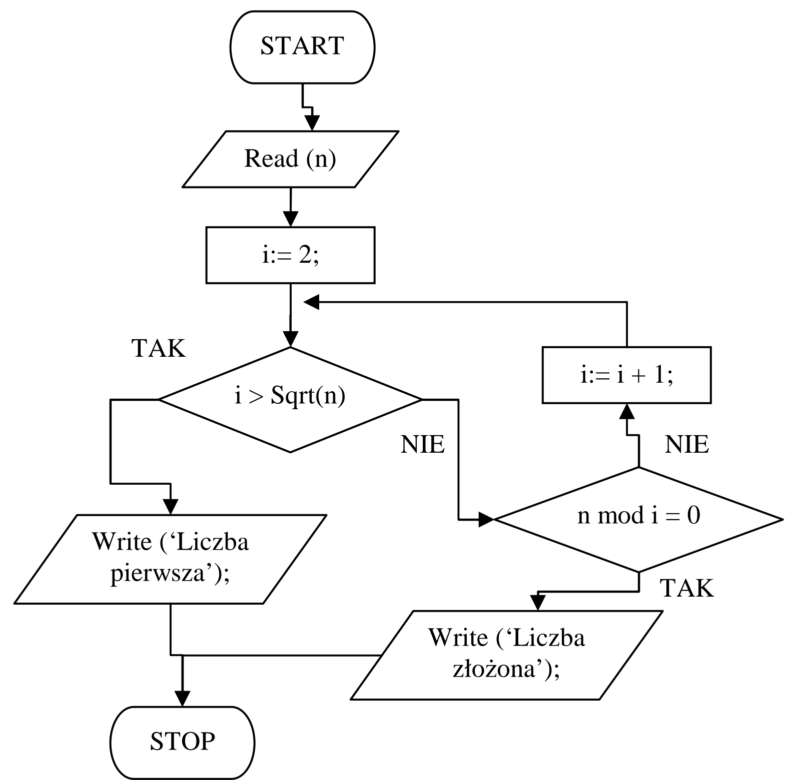
Algorytm testujący, czy wprowadzona liczba jest liczbą pierwszą

Największą liczbą pierwszą znalezioną 4 września 2006 jest liczba $2^{32,582,657} - 1$. w ramach projektu GIMPS (Great Internet Mersenne Prime Search – Wielkie Internetowe Poszukiwanie Liczb Pierwszych Mersenne'a ) (http://www.mersenne.org ) . Posiada ona 9,808,358 cyfr .
Duże liczby pierwsze służą do konstruowania szyfrów oraz wyszukiwania błędów w przekazie danych.

| | |
|---|---|
| Problem algorytmiczny : | badanie, czy wprowadzona liczba jest liczbą pierwszą |
| Dane wejściowe : | $n \in N$ , $n > 1$ – badana liczba |
| Dane wyjściowe : | napisz „liczba pierwsza" gdy n jest liczbą pierwszą lub „liczba złożona" gdy n nie jest liczbą pierwszą |
| Zmienne pomocnicze : | $i \in N$ – potencjalny podzielnik |

Aby to zbadać, będziemy liczbę „n" przez kolejne liczby naturalne mniejsze od „n", zaczynając od liczby 2 i sprawdzać resztę z dzielenia. Jeśli choć jeden dzielnik, dla którego reszta z dzielenia wynosi 0, to liczba n jest liczbą złożoną.
UWAGA : Jeśli liczba nie dzieli się przez żadną liczbę całkowitą równą jej pierwiastkowi kwadratowemu lub od niego mniejszą , to jest ona liczbą pierwszą.

***Schemat blokowy wyszukujący liczbę pierwszą.***
( dla naszego zadania poniższy schemat blokowy należy nieco zmodyfikować ).




Przykładowa procedura (do wczytania danych z pliku do tablicy należy wykorzystać funkcję Wczytaj_Dane_Z_Pliku_Do_Tablicy z poprzedniego zadania )

**ZADANIE 9**

Wygenerować wszystkie liczby pierwsze z zakresu (1, 50 000> i zapisać je do pliku „pierwsze.txt".

Wydawać by się mogło, że najprostszym sposobem jest iteracja liczb od 1 do 50 000 i sprawdzenie ich algorytmem z zadania 4.
Spróbujmy ułożyć program, który to zrobi.

Przykład programu „iteracyjnego".

Proszę zwrócić uwagę, że efektywność tego programu jest dosyć mała.
Spróbujmy zastosować trochę inny algorytm opierający się na właściwościach liczb pierwszych. Najprostszym sposobem uzyskania liczb pierwszych mniejszych od danej liczby (tu od 100 001) jest usunięcie ze zbioru wszystkich liczb podzielnych przez 2, 3 , 5 itp. aż do osiągnięcia liczby pierwszej, która spełnia nierówność $p^2 > N$. Pamiętajmy, że każda liczba złożona nie większa niż N ma dzielnik nie większy niż $\sqrt{N}$ . Algorytm taki nazywany jest **sitem Eratostenesa**.
Nazwa pochodzi od tego, że wszystkie liczby są po kolei przesiewane i usuwane są wszystkie wielokrotności danej liczby.

Przykład :

Znajdźmy na przykład wszystkie liczby pierwsze od 1 do 20. Liczby złożone będą po kolei "usuwane" w sposób podany niżej. Na początku nasz przedział wygląda następująco:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Cała idea tego algorytmu polega na tym, że "idziemy" od lewej strony przedziału poczynając od liczby 2 i gdy liczba ta nie została wcześniej skreślona to skreślamy wszystkie jej wielokrotności w danym przedziale **oprócz niej samej**.

Tak więc zaczynamy od liczby 2. Nie jest ona skreślona, więc skreślamy jej wielokrotności oprócz niej samej. W tym przypadku są to liczby: 4, 6, 8, 10, 12, 14, 16, 18, 20.

Czyli otrzymujemy zbiór :

1 2 3 5 7 9 11 13 15 17 19

Kolejną liczbę, jaką napotykamy po 2 jest liczba 3. Nie została ona wcześniej skreślona, więc skreślamy wszystkie jej wielokrotności w naszym przedziale od 1 do 20 oprócz niej samej. Proszę zauważyć, iż niektóre wielokrotności liczby 3 zostały już skreślone wcześniej. Liczby, które w tym przypadku skreślimy to: 9, 15 (6, 12 i 18 już wcześniej skreśliliśmy). Po tej operacji otrzymamy

1 2 3 5 7 11 13 17 19

Kolejną liczbę jaką bierzemy to 5. Ale proszę zwrócić uwagę, że ograniczyliśmy zbiór do 20, a pierwiastek z 20 to około 4,4. Pamiętajmy, że każda liczba złożona nie większa niż N ma dzielnik nie większy niż $\sqrt{N}$ .Stąd nie musimy już rozpatrywac liczby 5. Ktoś może powiedzieć, że 20 dzieli się przez 5 wobec tego musimy ją wziąć pod uwagę. Ale każda liczba złożona w danym przedziale to iloczyn dwóch innych liczb (nie muszą być one koniecznie pierwsze). Liczby te mogą być równe sobie - wtedy wartość tych liczb to pierwiastek kwadratowy z danej liczby złożonej. Jednakże, gdy liczby te nie są sobie równe - to jedna jest większa, a druga mniejsza. Mniejsza z tych liczb jest mniejsza od pierwiastka kwadratowego z danej liczby złożonej. Tak więc, aby skreślić daną liczbę

złożoną musimy dojść (gdy idziemy po kolei od 2) do tej mniejszej liczby, nie musimy wcale dochodzić do tej większej.

Dane wejściowe : g – górna granica zbioru (pamiętajmy ! $g \in N$ )
Dane wyjściowe : kolejne liczby pierwsze z przedziału od 2 do g
Zmienne pomocnicze : i – sterowanie iteracjami ,
w – służy do tworzenia wielokrotności
tablica [] - tablica logiczna odwzorowująca zbiór liczbowy. Indeksy przebiegają wartości od 2 do g

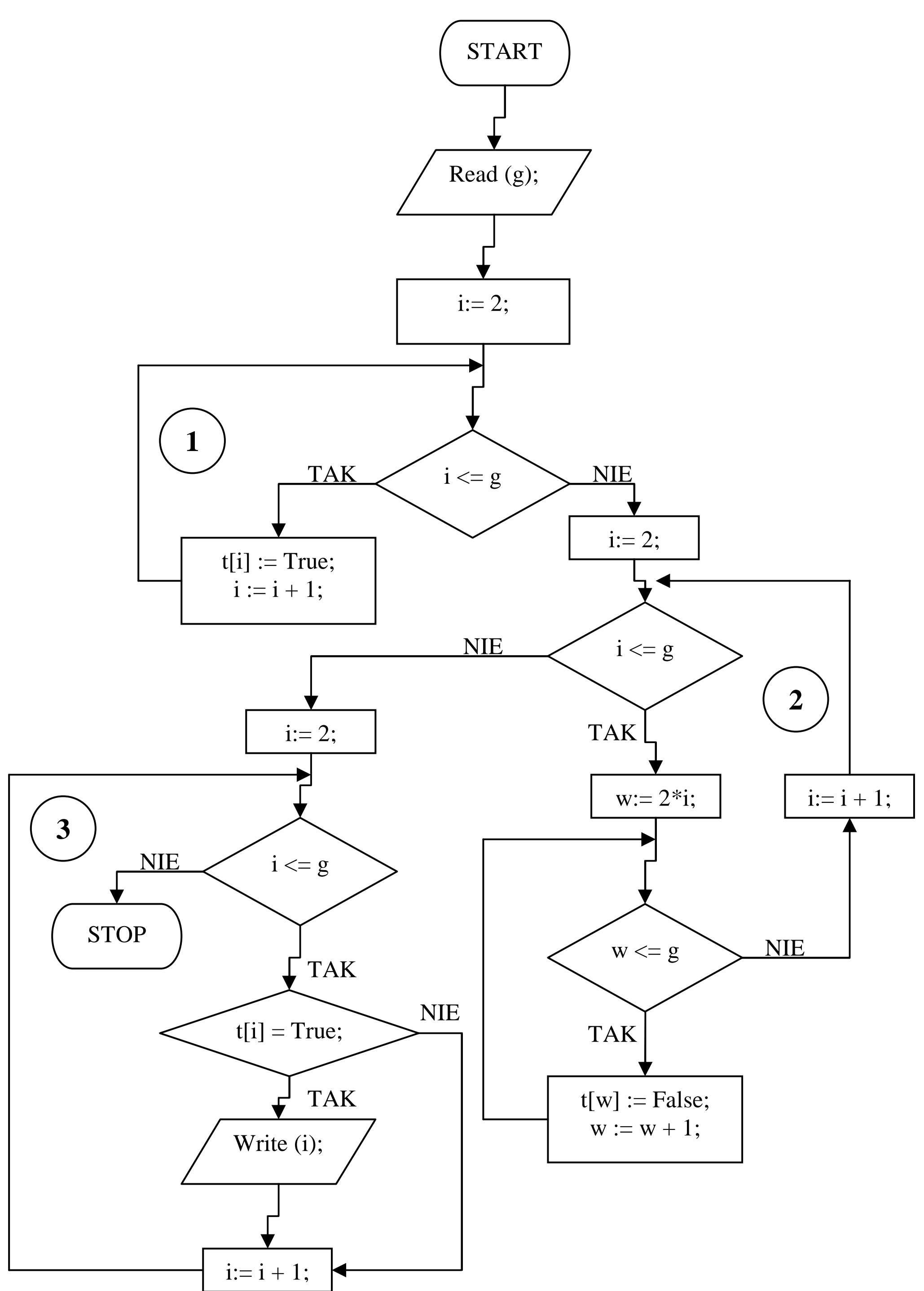
| 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|
| True | True | Flase | True | False |

***Algorytm w postaci listy kroków :***

Krok 1 : podanie g
Krok 2 : dla i=2 , 3, ....g wykonaj t[i] := True; {wstawienie, że wszystkie elementy tablicy są liczbami pierwszymi }
Krok 3 : dla i = 2, 3, .....g wykonuj kroki 4, 5, 6, 7
Krok 4 : podstaw w := 2i;
Krok 5 : Dopóki w <= g wykonaj kroki 6, 7. Jeśli nie to wykonaj następny obieg petli z kroku 3
Krok 6 : podstaw t[w] := False; { liczba jest złożona }
Krok 7 : podstaw w := w + 1;
Krok 8 : dla i = 2, 3, .... g jeżeli t[i] = True to pisz i { wypisuj liczby pierwsze }
Krok 9 : koniec

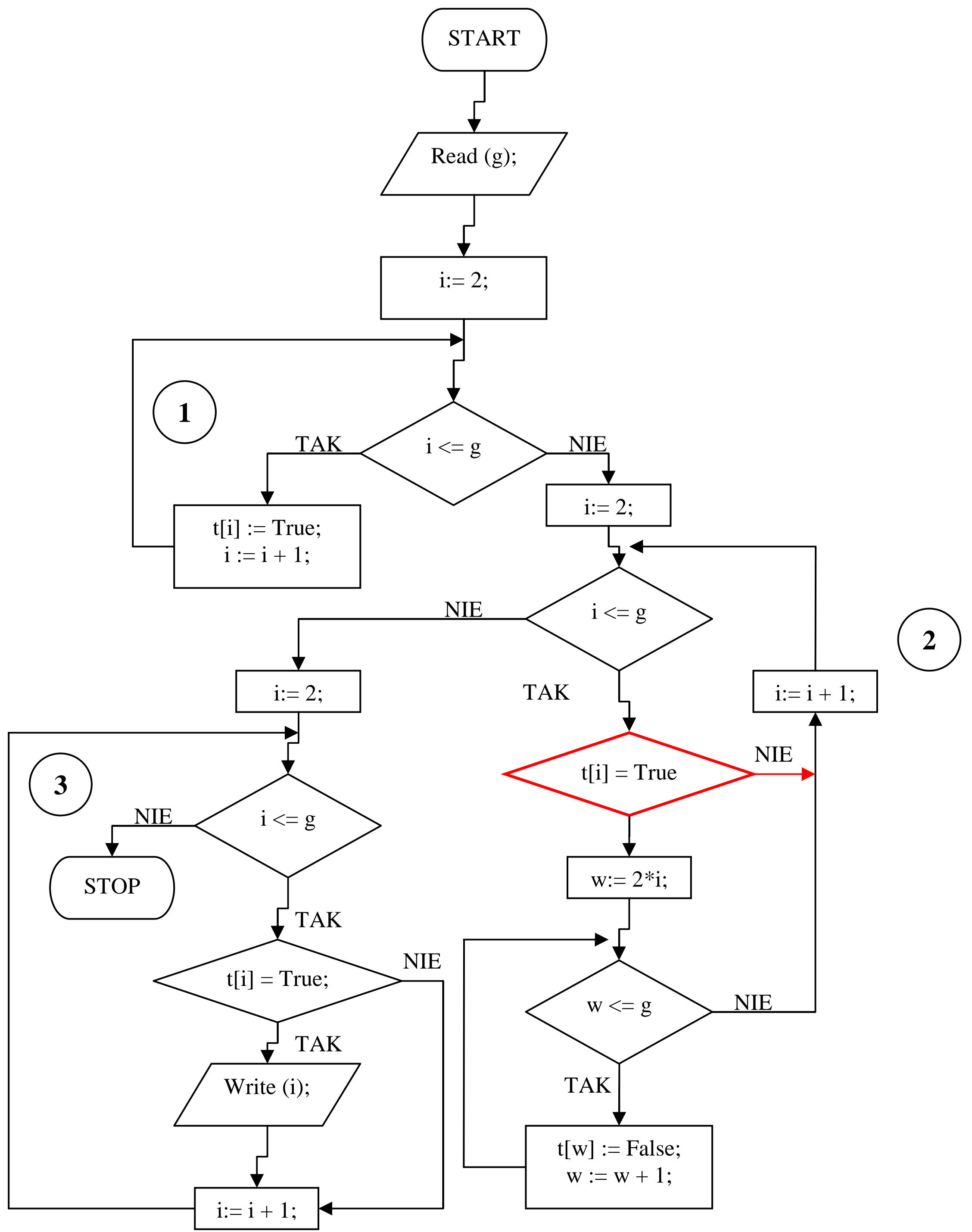
***Algorytm w schematu blokowego - wersja 1:***

Pętla 1 – ustawia wszystkie elementy tablicy na True. Tablica ustawiana jest od indeksu 2. Pamiętajmy, że tablica ta jest odzwierciedleniem liczb naturalnych, przy czym wartości są w niej zawarte jako indeksy, a elementy tablicy przyjmują wartości True (liczba pierwsza) lub False (liczba złożona).
Pętla 2 - usuwa ze zbioru wszystkie wielokrotności klejnych liczb.
Pętla 3 - wyświetla wszystkie elementy tablicy, a dokładniej indeksy, których wartości są True.




Jeśli przyjrzymy się dokładnie pętli nr 2, dojdziemy do wniosku, iż pewne wielokrotności liczb są wyrzucane ze zbioru kilka razy. Np. dla liczb 2, 4, 6 itp. Możemy znacznie zmniejszyć liczbę operacji wyrzucania, jeśli na samym początku sprawdzimy, czy liczba, której wielokrotności mają być wyeliminowane ze zbioru, faktycznie w tym zbiorze występuje. Jeśli jej nie ma (gdyż została już wcześniej wyrzucona), to oczywiście nie może także być żadnej jej wielokrotności.

***Algorytm w schematu blokowego - wersja 2:***

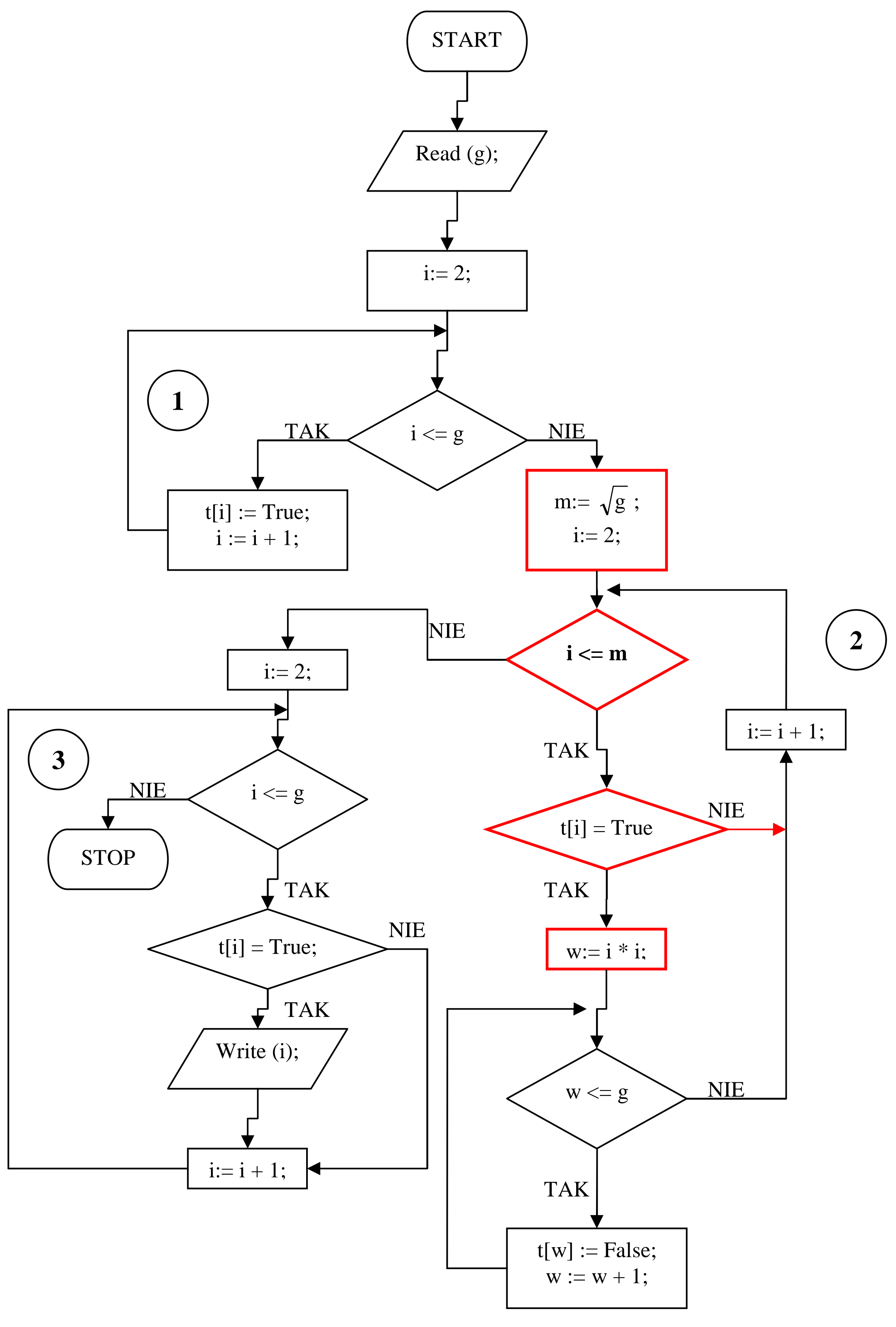



Wprowadzona zmiana sprawdza, czy liczba reprezentowana przez element tablicy t[i] jest w zbiorze (ma wartość logiczną T*rue*). Jeśli tak, algorytm usuwa z tego zbioru wszystkie jej wielokrotności. Jeśli liczby w zbiorze nie ma, następuje przejście na koniec pętli i kontynuacja z kolejną liczbą.
Zwróćmy też uwagę, że nie musimy przeglądać całego zbioru.
W pierwszym obiegu pętli nr 2 algorytm usuwa ze zbioru wszystkie wielokrotności liczby 2. W drugim obiegu algorytm natrafia na liczbę 3. Pierwsza wielokrotność 3 jest równa 6 i została usunięta już w poprzednim obiegu, ponieważ 6 ma czynnik 2 (dzieli się bez reszty przez 2). Pierwszą nieusuniętą wielokrotnością będzie kwadrat 3, czyli 9. W kolejnym obiegu liczba 4

zostanie pominięta, ponieważ usunięto ją już ze zbioru. Dalej algorytm napotka liczbę 5. Pierwszą nieusuniętą wielokrotnością 5 jest znów kwadrat 5, ponieważ:
10 = 2 * 5 - usunięte przy 2
15 = 3 * 5 - usunięte przy 3
20 = 2 * 2 * 5 - usunięte przy 2

To nie jest przypadek, iż przy 3 i 5 otrzymujemy jako pierwszą istniejącą wielokrotność kwadrat danej liczby pierwszej. Tak samo będzie z kolejnymi liczbami pierwszymi, ponieważ ich wielokrotności mniejsze od kwadratu liczby posiadają dzielniki równe wcześniej znalezionym liczbom pierwszym, zatem zostały usunięte już ze zbioru przy okazji usuwania wielokrotności tych liczb. Wynika stąd oczywisty wniosek:
**Wielokrotności liczb większych od pierwiastka górnej granicy zbioru są usunięte ze zbioru w trakcie usuwania wielokrotności ich czynników pierwszych. Zatem przeglądanie zbioru można przerwać po przekroczeniu granicy równej pierwiastkowi kwadratowemu górnej granicy zbioru.**
Czyli zamiast przeglądać zbiór liczbowy od 2 do 50 000 wystarczy wyrzucić wszystkie wielokrotności liczb od 2 do 223 ( $\sqrt{50000} \approx 223{,}6$ ). Oszczędność oczywista – około 200 razy mniej liczb do przeglądnięcia.
Następne ulepszenie wiąże się z wyznaczeniem pierwszej wielokrotności do usunięcia. Z powyższych rozważań wynika, iż w trakcie działania algorytmu Sita Eratostenesa ze zbioru są wcześniej wyrzucane wielokrotności mniejsze od kwadratu liczby. Zatem przyjmiemy za pierwszą wielokrotność właśnie kwadrat liczby - mniejszych wielokrotności nie trzeba wyrzucać, bo już zostały wyrzucone. W efekcie znów obniżymy ilość niezbędnych operacji.

***Algorytm w schematu blokowego - wersja ostateczna:***



***Przykładowy program :***

```
Program Sito_Eratostenesa;
Uses Crt;
Const g=50000;
var
  tablica   : array[2..g] of boolean;
  i,w,licznik,m     : LongInt;
  plik_1 : Text;

    {*******************************************************}
    Procedure Zapisz_Do_Pliku (liczba : LongInt);
    Var l      : String;
    Begin
       Str (liczba,l);
       Writeln (plik_1,l);
       Writeln (l);
    End;
    {*******************************************************}

Begin
    ClrScr;
    licznik := 0;
    Assign (plik_1,'pierwsze.txt');
    Rewrite (plik_1);
    For i := 2 To g Do tablica[i] := true;
    m:= Round (Sqrt(g));
    For i := 2 To m Do
    Begin
            If tablica [i]  = True Then
            Begin
                    w := i * i;
                    While w <= g Do
                    Begin
                         tablica [w] := False;
                         w    := w + i;
                     End;
            End;
   End;

    For i := 2 To g Do If tablica[i] then
                Begin
                    Zapisz_Do_Pliku (i);
                    licznik:=licznik+1;
                End;
    Writeln ('W znaleziono i zapisano ',licznik,' liczb pierwszych');
    Close (plik_1);
    Repeat Until KeyPressed;
End.
```

Liczb pierwszych jest 5133. Program pierwszy wykonuje się prawie o 1/3 dłużej niż program 2.

# 8. Sortowanie zbioru.

## ZADANIE 10

Ułożyć wszystkie liczby znajdujące się w pliku „liczby.txt” od najmniejszej do największej, a wynik zapisać w pliku „sort.txt”.

Sortowanie danych jest jednym z podstawowych problemów programowania komputerów, z którym prędzej czy później spotka się każdy programista. Poniżej przedstawiamy tylko nieliczne dziedziny, w których występuje potrzeba sortowania danych:

- sport - wyniki uzyskane przez poszczególnych zawodników należy ułożyć w określonej kolejności, aby wyłonić zwycięzcę oraz podać lokatę każdego zawodnika.
- bank - spłaty kredytów należy ułożyć w odpowiedniej kolejności, aby wiadomo było kto i kiedy ma płacić odsetki do banku.
- grafika - wiele algorytmów graficznych wymaga porządkowania elementów, np. ścian obiektów ze względu na odległość od obserwatora. Uporządkowanie takie pozwala później określić, które ze ścian są zakrywane przez inne ściany dając w efekcie obraz trójwymiarowy.
- bazy danych - informacja przechowywana w bazie danych może wymagać różnego rodzaju uporządkowania, np. lista książek może być alfabetycznie porządkowana wg autorów lub tytułów, co znacznie ułatwia znalezienie określonej pozycji.

Wbrew obiegowym opiniom nie ma "idealnego" i "uniwersalnego" algorytmu sortującego - jeden algorytm jest lepszy w jednej sytuacji, drugi w innej. Dlatego dokładna znajomość własności algorytmów sortujących pozwala na tworzenie naprawdę efektywnych aplikacji.

Z problemem sortowania powiązane są problemy wyszukiwania danych ponieważ duża klasa algorytmów wyszukujących operuje na zbiorach posortowanych.
W informatyce **algorytm wyszukujący** otrzymuje na wejściu pewien problem i daje na wyjściu jego rozwiązanie po przetestowaniu pewnej ilości możliwych rozwiązań.
Większość algorytmów rozwiązujących problemy (np. algorytmy gry w szachy, warcaby, karty itp.) są pewnego rodzaju algorytmami wyszukującymi. Zbiór wszystkich możliwych rozwiązań danego problemu nosi nazwę **przestrzeni poszukiwań** (ang. search space). Najprostszymi algorytmami wyszukującymi są **algorytmy naiwne**, które stosują najbardziej intuicyjną metodę wyszukiwania w przestrzeni poszukiwań. Wymyślono jednakże bardziej zaawansowane metody wykorzystujące wiedzę na temat samej struktury przestrzeni poszukiwań w celu zmniejszenia ilości czasu traconego na poszukiwania.

Przejdźmy jednak do sortowania.

**Zbiór posortowany** to taki, w którym kolejne elementy są poukładane w pewnym porządku (kolejności). Porządek ten możemy określić za pomocą relacji >= lub <= (albo dowolnej innej relacji porządkowej, która jednoznacznie wyznacza kolejność elementów w zbiorze). Równość jest konieczna w przypadku, gdy elementy zbioru mogą przyjmować takie same wartości.
Na przykład zbiór liczb: D = { 1, 3, 3, 5, 7, 7, 8, 9 }
jest posortowany rosnąco, ponieważ pomiędzy każdymi dwoma kolejnymi elementami zachodzi relacja: element poprzedni jest mniejszy lub równy od elementu następnego.

Odwrotnie, zbiór: D = { 9, 8, 6, 6, 4, 2, 1, 1, 0 }

jest posortowany malejąco, ponieważ pomiędzy każdymi dwoma kolejnymi elementami zachodzi relacja: element poprzedni jest większy lub równy od elementu następnego.

Oczywiście zbiór wcale nie musi składać się z liczb (chociaż tak naprawdę każdy rodzaj danych komputerowych w końcu sprowadza się do liczb, gdyż wewnętrznie jest reprezentowany przez kody binarne – o tym powiemy sobie później). W takim przypadku należy określić dla każdego elementu tzw. klucz (ang. key), wg którego dokonywane jest sortowanie. Ponieważ klucz jest liczbą, zatem obowiązują dla niego podane wyżej zasady kolejności elementów.
Na przykład dla tekstów kluczem mogą być kody poszczególnych znaków. Większość języków programowania posiada operatory porównywania ciągu znaków (problemem może być sortowanie wg zasad języka polskiego - nie wystarczy wtedy porównywać kodów znakowych, ponieważ kody polskich literek ą, Ą, ć, Ć itd. są zwykle większe od kodów liter alfabetu łacińskiego, ale i ten problem daje się z powodzeniem rozwiązać przez odpowiedni dobór kluczy).

Tak więc przez sortowanie będziemy rozumieć taką zmianę kolejności elementów zbioru nieuporządkowanego (permutację), aby w wyniku spełniały one założony porządek.

Kolejnym zagadnieniem, które powinniśmy omówić we wstępie, jest tzw. **czasowa złożoność obliczeniowa** (ang. computational complexity) algorytmu sortującego (istnieje również złożoność pamięciowa). Określa ona statystycznie czas wykonywania algorytmu w zależności od liczby danych wejściowych. Czasowa złożoność obliczeniowa wyrażana jest liczbą tzw. operacji dominujących, czyli takich, które mają bezpośredni wpływ na czas wykonywania algorytmu. Dzięki takiemu podejściu uniezależniamy czasową złożoność obliczeniową od szybkości komputera, na którym dany algorytm jest realizowany. Złożoność obliczeniową charakteryzujemy przy pomocy tzw. notacji Θ(omikron). Oto odpowiednie przykłady:

$\Theta(n)$ — Algorytm o liniowej zależności czasu wykonania od ilości danych. Dwukrotny wzrost liczby przetwarzanych danych powoduje dwukrotny wzrost czasu wykonania. Tego typu złożoność powstaje, gdy dla każdego elementu należy wykonać stałą liczbę operacji.

$\Theta(n^2)$ — Algorytm, w którym czas wykonania rośnie z kwadratem liczby przetwarzanych elementów. Dwukrotny wzrost liczby danych powoduje czterokrotny wzrost czasu wykonania. Tego typu złożoność powstaje, gdy dla każdego elementu należy wykonać ilość operacji proporcjonalną do liczby wszystkich elementów.

$\Theta(n \log n)$ — Dobre algorytmy sortujące mają taką właśnie złożoność obliczeniową. Czas wykonania przyrasta dużo wolniej od wzrostu kwadratowego. Tego typu złożoność powstaje, gdy zadanie dla n elementów można rozłożyć na dwa zadania zawierające po połowie elementów.

$\Theta(n!)$
$\Theta(a^n)$ — Bardzo pesymistyczne algorytmy, czas wykonania rośnie szybko ze wzrostem liczby elementów wejściowych, czyli znalezienie rozwiązania może zająć najszybszym komputerom całe wieki lub tysiąclecia. Takich algorytmów należy unikać jak ognia !

Zapis Θ ( ) określamy mianem klasy złożoności obliczeniowej algorytmu. Klasa czasowej złożoności obliczeniowej umożliwia porównywanie wydajności różnych algorytmów sortujących. Z reguły proste algorytmy posiadają wysoką złożoność obliczeniową - długo dochodzą do wyniku końcowego. Algorytmy bardziej skomplikowane posiadają mniejszą złożoność obliczeniową -

szybko dochodzą do rozwiązania. Złożoność obliczeniowa wszystkich algorytmów sortujących została dokładnie oszacowana co również uwzględnimy.

| Porównanie klas złożoności obliczeniowych | | |
|---|---|---|
| **Klasa złożoności obliczeniowej** | **Nazwa klasy złożoności obliczeniowej** | **Cechy algorytmu** |
| $\Theta(1)$ | stała | działa prawie natychmiast |
| $\Theta$ (log n) | logarytmiczna | bardzo szybki |
| $\Theta$ (n) | liniowa | szybki |
| $\Theta$ (n log n) | liniowo-logarytmiczna | dosyć szybki |
| $\Theta$ ($n^2$) | kwadratowa | wolny dla dużych n |
| $\Theta$ ($n^3$) | sześcienna | wolny dla większych n |
| $\Theta$ ($2^n$), $\Theta$ (n!) | wykładnicza | nierealizowalny dla większych n |

Nieformalnie mówiąc klasa złożoność obliczeniowej informuje nas, iż czas t w funkcji liczby elementów n jest proporcjonalny do funkcji określonej przez klasę złożoności, a stała c jest współczynnikiem tej proporcjonalności. Spostrzeżenie to pozwala w przybliżeniu oszacować czas wykonania algorytmu dla n elementów, jeśli jest znana jego klasa złożoności obliczeniowej oraz czas wykonania dla innej liczby elementów.

Przykład.
Załóżmy, iż mamy program sortujący zbudowany na bazie algorytmu sortującego o klasie złożoności obliczeniowej $\Theta$ ($n^2$). Sto elementów jest sortowane w czasie 1 sekundy. Ile czasu zajmie posortowanie za pomocą tego programu zbioru o tysiącu elementach ? Odpowiedź brzmi - 100 sekund. Ponieważ ilość danych wzrosła 10 razy, to czas obliczeń wzrósł $10^2$, czyli 100 razy. Poniżej przedstawiamy odpowiednią tabelkę.

| Lp. | n | Czas obliczeń |
|---|---:|---|
| 1. | 100 | = 1 sekunda |
| 2. | 1.000 | = 100 sekund = 1 minuta 40 sekund |
| 3. | 10.000 | = 10.000 sekund = 2 godziny 46 minut 40 sekund |
| 4. | 100.000 | = 1.000.000 sekund = 11 dni 13 godzin 46 minut 40 sekund |
| 5. | 1.000.000 | = 100.000.000 sekund = 3 lata 2 miesiące 9 godzin 46 minut 40 sekund |
| 6. | 10.000.000 | = $1 \times 10^{10}$ sekund = 317 lat 1 miesiąc 4 dni 17 godzin 46 minut 40 sekund |

Widzimy więc, iż algorytm ten spisuje się dobrze tylko przy niewielkiej liczbie elementów. Gdy liczba sortowanych elementów jest duża, czas oczekiwania na rozwiązanie może być nie do zaakceptowania. Dlatego właśnie informatycy poświęcili tak dużo swojego wysiłku na opracowanie odpowiednio szybkich algorytmów sortujących (te najszybsze mają klasę złożoności $\Theta$ (n log n) ).

Oprócz złożoności czasowej rozważa się również **złożoność pamięciową**. Określa ona ilość zasobów komputera, których wymaga dany algorytm w zależności od liczby danych wejściowych. Tutaj także ma zastosowanie notacja omikron. Przy określaniu złożoności algorytmu należy wziąć pod uwagę oba typy złożoności obliczeniowej.

Ze względu na złożoność pamięciową algorytmy sortujące dzielimy na dwie podstawowe grupy:

- Algorytmy sortujące w miejscu (ang. in place) - wymagają stałej liczby dodatkowych struktur danych, która nie zależy od liczby elementów sortowanego zbioru danych (ani od ich wartości). Dodatkowa złożoność pamięciowa jest zatem klasy $\Theta(1)$. Sortowanie odbywa się wewnątrz zbioru. Ma to bardzo istotne znaczenie w przypadku dużych zbiorów danych, gdyż mogłoby się okazać, iż posortowanie ich nie jest możliwe z uwagi na brak pamięci w systemie. Większość opisanych tu algorytmów sortuje w miejscu, co jest ich bardzo dużą zaletą.

- Algorytmy nie sortujące w miejscu - wymagają zarezerwowania w pamięci dodatkowych obszarów, których wielkość jest uzależniona od liczby sortowanych elementów lub od ich wartości. Tego typu algorytmy są zwykle bardzo szybkie w działaniu, jednakże okupione to jest dodatkowymi wymaganiami na pamięć. Zatem w pewnych sytuacjach może się okazać, iż taki algorytm nie będzie w stanie posortować dużego zbioru danych, ponieważ system komputerowy nie posiada wystarczającej ilości pamięci RAM.

Algorytmy sortujące dzieli się również na dwie grupy:

- Algorytmy stabilne - zachowują kolejność elementów równych. Oznacza to, iż elementy o tych samych wartościach będą występowały w tej samej kolejności w zbiorze posortowanym, co w zbiorze nieposortowanym. Czasami ma to znaczenie, gdy sortujemy rekordy bazy danych i nie chcemy, aby rekordy o tym samym kluczu zmieniały względem siebie położenie.

- Algorytmy niestabilne - kolejność wynikowa elementów równych jest nieokreślona (zwykle nie zostaje zachowana).

Przejdźmy teraz do omówienia algorytmów sortujących.

## *8.1. Sortowanie zwariowane (Bogo Sort)*

Pierwszy z prezentowanych algorytmów sortujących opiera się na dosyć zwariowanych zasadach. Jego działanie możemy scharakteryzować na przykładzie układania talii kart. Bierzemy talię kart. Sprawdzamy czy jest ułożona. Jeśli nie, tasujemy ją i znów sprawdzamy ułożenie. Operacje sprawdzania i tasowania wykonujemy dotąd, aż talia nam się ułoży w pożądanej kolejności kart.

Nic nie sortujemy, wręcz dokonujemy operacji odwrotnej - tasowania, a talia może zostać posortowana. Dlaczego? Wynika to z praw rachunku prawdopodobieństwa. Otóż tasowanie powoduje, iż karty przyjmują losowe permutacje swoich położeń. Ponieważ każda permutacja zbioru kart jest równie prawdopodobna (jeśli przy tasowaniu nie oszukujemy), zatem możemy też otrzymać układ uporządkowany. Oczywiście wynik taki pojawia się dosyć rzadko (bądźmy szczerzy - przy dużej liczbie elementów bardzo, bardzo... rzadko). Nie poleca się sortowania tą metodą zbiorów liczniejszych niż 9 elementów.

Algorytm opiera się na losowym sortowaniu zbioru. Tymczasem w komputerze nie mamy tak naprawdę dostępu do liczb czysto losowych. Zadowalamy się ich przybliżeniem, czyli liczbami pseudolosowymi powstającymi na bazie algorytmicznej. Może się zatem zdarzyć, iż nasz generator pseudolosowy nigdy nie wygeneruje potrzebnej sekwencji liczb pseudolosowych, zatem algorytm sortujący nie będzie w stanie ukończyć swojej pracy.

Z tego powodu jest to jeden z najgorszych algorytmów sortujących. Posiada pesymistyczną czasową złożoność obliczeniową klasy $\Theta(n \times n!)$. Złożoność taką nazywamy złożonością super wykładniczą. Co gorsze, ten sam zbiór raz może zostać błyskawicznie posortowany (gdy akurat

mamy szczęście) a innym razem możemy czekać na wynik nawet cały rok (albo jeszcze dłużej). Sortowanie odbywa się w miejscu.

***Szczegóły implementacyjne algorytmu.***

Jeśli za pomocą podanego algorytmu chcemy posortować zbiór liczbowy, to musimy rozwiązać dwa istotne problemy:

- **Sprawdzenie posortowania elementów**. Aby upewnić się, iż zbiór jest posortowany, należy porównać ze sobą wszystkie kolejne sąsiednie elementy. Jeśli spełniają założoną kolejność, to zbiór jest uporządkowany. Jeśli chociaż jedna para elementów zbioru jest w złej kolejności ze względu na przyjęty porządek, to zbiór nie jest uporządkowany
- **Losowe potasowanie**. Operacja ta ma na celu pomieszanie elementów w zbiorze, aby przyjęły przypadkowe pozycje. Najprościej dokonamy tego losując dwa numery elementów, a następnie zamieniając wylosowane elementy miejscami. Jeśli operację taką powtórzymy wystarczającą liczbę razy (np. 3-krotną liczbę elementów w zbiorze), to zawartość zbioru zostanie potasowana (wymieszana).
  Podstawową operacją jest zamiana zawartości dwóch elementów zbioru. Wymaga ona trzech kroków oraz zmiennej pomocniczej do tymczasowego przechowania jednego z elementów. W pierwszym kroku przenosimy do zmiennej pomocniczej jeden z elementów. W drugim kroku na zwolnione miejsce wstawiamy drugi z elementów, a na koniec w kroku trzecim na miejsce drugiego elementu przenosimy element zapamiętany w zmiennej pomocniczej.
  x:= a;    a:= b;  b:= x;
  Kolejna operacja to losowanie indeksu elementu. Wykorzystamy tutaj generator liczb pseudolosowych. Wygenerowany indeks powinien być liczbą pseudolosową z zakresu od 1 do n, gdzie n oznacza ilość elementów w zbiorze. Operację tę wykonujemy następująco:
  indeks := 1 + random (n);

***Specyfikacja algorytmu:***

*Dane wejściowe :*
n - liczba elementów w sortowanym zbiorze, $n \in N$
d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do n.

*Dane wyjściowe :*
d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do n.

*Zmienne pomocnicze :*
i - zmienna sterująca pętli, $i \in N$
i1, i2 - losowe indeksy elementów zbioru d[ ], $i1, i2 \in N$

***Lista kroków***

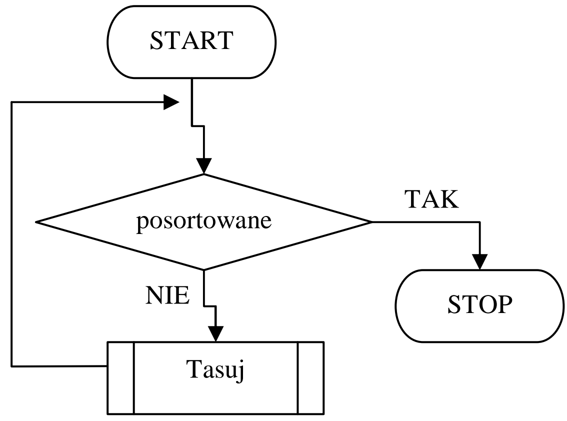
Algorytm główny :
Krok 1 : Dopóki **Posortowane** = false to **Tasuj**
Krok 1 : Zakończ algorytm

Algorytm funkcji Posortowane
Krok 1 : Dla i = 1,2,...,n - 1, jeśli d[i] > d[i + 1], to *Posortowane* := false i zakończ algorytm
Krok 2 : *Posortowane* := true i zakończ algorytm

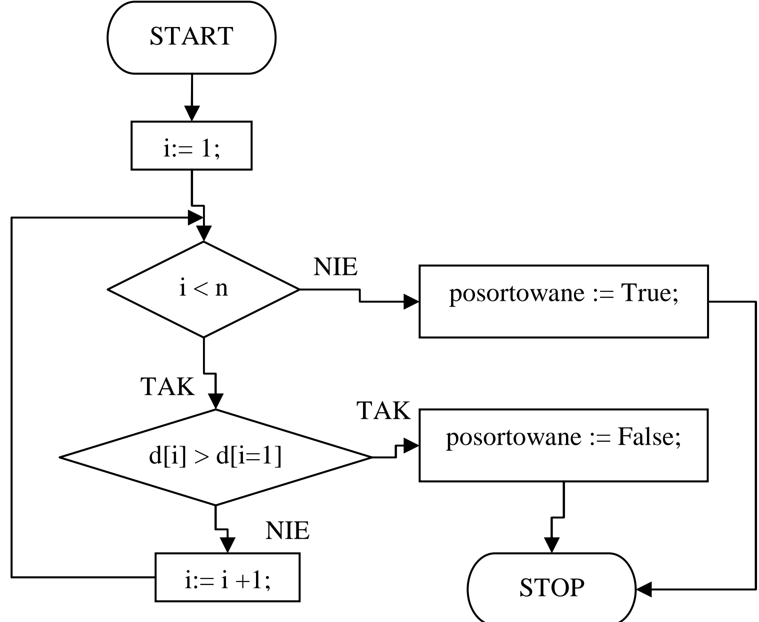
Algorytm procedury Tasuj
Krok 1 : Dla i = (1,2,...,3) * n, wykonuj kroki 2 i 3
Krok 2 : Wylosuj indeksy i1 i i2 w przedziale od 1 do n
Krok 3 : Wymień zawartości d [i1] ↔ d [i2]
Krok 4 : Zakończ algorytm

***Schemat blokowy***

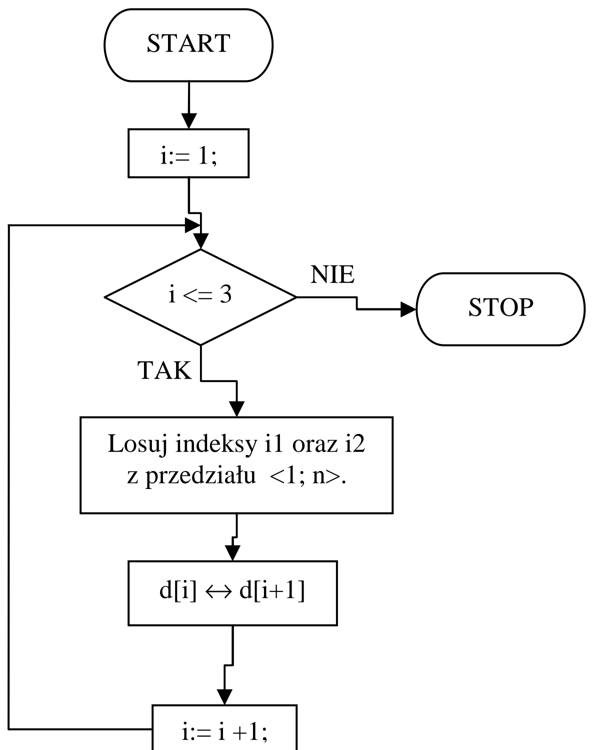
Algorytm główny




Algorytm funkcji Posortowane



Algorytm procedury Tasuj




Przykładowy program.

```
Program Sortowanie_Zwariowane;
Uses Crt;
Const N = 8; { Liczebnosc zbioru. Nie wstawiaj liczb od 9 !!! }
Var  d : array[1..N] of Integer;

   { Funkcja Posortowane - sprawdzajaca uporzadkowanie w zbiorze }
   {*********************************************************}
   Function Posortowane : boolean;
   Var  i : integer;
   Begin
       For i := 1 To N - 1 Do
        If d[i] > d[i+1] Then
        Begin
            Posortowane := False;
            Exit;
        End;
        Posortowane := True;
   End;
   {*********************************************************}
   { Procedura tasujaca zbior }
   Procedure Tasuj;
   Var
     i,i1,i2,x : Integer;
   Begin
       For i := 1 To 3*N Do
       Begin
           i1 := 1 + Random(N);
```

```
      i2 := 1 + Random(N);
      x := d[i1];
      d[i1] := d[i2];
      d[i2] := x;
    End;
  End;

  {*******************************************************}
Var   i : Integer;
Begin
  ClrScr;
  Writeln('Sortowanie  zwariowane');
  Writeln;
{ Najpierw wypelniamy tablice d[] liczbami pseudolosowymi, a nastepnie wyswietlamy jej zawartosc }
  Randomize;
  For i:=1 To N Do d[i] := Random(10000);
  Writeln('Przed sortowaniem:');
  Writeln;
  For i:=1 To N Do Write(d[i] : 6);
  Writeln; Writeln;

{ Sortowanie }      While not Posortowane Do Tasuj;

{ Wynik sortowania }
  Writeln('Po sortowaniu:');
  Writeln;
  For i:=1 To N Do Write(d[i] : 6);
  Writeln; Writeln;
  Writeln ('Sortowanie zakonczone - nacisnij dowolny klawisz');
  Repeat Until KeyPressed; {oczekiwanie na nacisniecie klawisza}
End.
```

Zaprezentowany algorytm jest ekstremalnie złym algorytmem sortującym i na pewno nie należy go stosować !! Został o tu zaprezentowany tylko w celu pokazania procedur (funkcji) służących sprawdzeniu posortowania oraz tasowania elementów zbioru.

Jako ciekawostkę podam fakt, iż informatycy terminem „bogo sort" określają program lub algorytm, którego idea działania jest tak beznadziejnie głupia, iż praktycznie nie może dać rozwiązania w sensownym okresie czasu. Zatem jeśli usłyszysz zdanie: "twój program to bogo sort", to już będziesz wiedział o co chodzi rozmówcy... :)

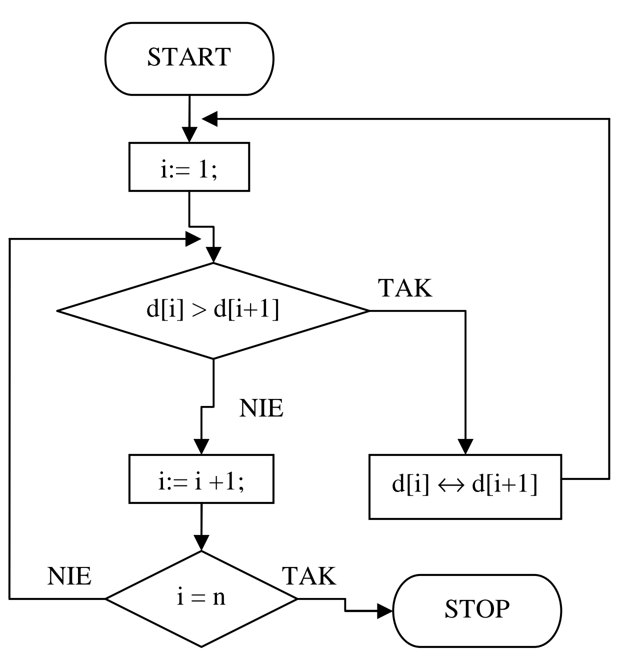
| **Cechy Algorytmu Sortowania Zwariowanego** | |
|---|---|
| Klasa złożoności obliczeniowej | Θ(n x n!) |
| Sortowanie w miejscu | TAK |
| Stabilność | NIE |

## *8.2. Sortowanie naiwne (głupie).*

Sortowanie naiwne jest również bardzo złym algorytmem sortującym, lecz, w przeciwieństwie do opisanego w poprzednim rozdziale sortowania zwariowanego, daje zawsze poprawne wyniki.
Zasada działania jest bardzo prosta:
Przeglądamy kolejne pary sąsiednich elementów sortowanego zbioru. Jeśli bieżąco przeglądana para elementów jest w złej kolejności, elementy pary zamieniamy miejscami i całą operację rozpoczynamy od początku zbioru. Jeśli przeglądniemy wszystkie pary, zbiór będzie posortowany.
Naiwność algorytmu wyraża się tym, iż po napotkaniu nieposortowanych elementów algorytm zamienia je miejscami, a następnie rozpoczyna całą pracę od początku zbioru. Złożoność obliczeniowa algorytmu przy sortowaniu zbioru nieuporządkowanego ma klasę Θ(n3). Sortowanie odbywa się w miejscu.

Algorytm sortowania naiwnego występuje w dwóch wersjach - rekurencyjnej oraz iteracyjnej. Wersja rekurencyjna jest jeszcze gorsza od iteracyjnej, gdyż dodatkowo zajmuje pamięć na kolejne poziomy wywołań rekurencyjnych, dlatego nie będziemy się nią zajmować

***Schemat blokowy :***




Rozpoczynamy przeglądanie zbioru od pierwszego elementu - indeks i przyjmuje wartość 1. W pętli sprawdzamy kolejność elementu d[i] z elementem następnym d[i+1]. Ponieważ założyliśmy porządek rosnący, to w przypadku d[i] > d[i+1] elementy te są w złej kolejności (dla porządku malejącego należy zmienić relację większości na relację mniejszości). W takiej sytuacji zamieniamy miejscami elementy, indeks i ustawiamy z powrotem na 1 (powrót na sam początek algorytmu byłby nieco kłopotliwy do zrealizowania w językach programowania, stąd dla prostoty ustawiamy i na 1 za operacją zamiany elementów) i wracamy na początek pętli.

Jeśli porównywane elementy są w dobrej kolejności, zwiększamy indeks i o 1, sprawdzamy, czy osiągnął już wartość końcową n i jeśli nie, wracamy na początek pętli. W przeciwnym razie kończymy - zbiór jest posortowany.

| **Cechy Algorytmu Sortowania Naiwnego** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta(n)$ - $\Theta(n^2)$ |
| klasa złożoności obliczeniowej typowa | $\Theta(n^3)$ |
| klasa złożoności obliczeniowej pesymistyczna | $\Theta(n^3)$ |
| Sortowanie w miejscu | TAK |
| Stabilność | TAK |

Klasy złożoności obliczeniowej szacujemy następująco:

- optymistyczna - dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoich miejscach)
- typowa - dla zbiorów o losowym rozkładzie elementów
- pesymistyczna - dla zbiorów posortowanych odwrotnie

## *8.3. Sortowanie bąbelkowe.*

Algorytm sortowania bąbelkowego jest jednym z najstarszych algorytmów sortujących. Można go potraktować jako ulepszenie opisanego w poprzednim rozdziale algorytmu sortowania naiwnego. Zasada działania opiera się na cyklicznym porównywaniu par sąsiadujących elementów i zamianie ich kolejności w przypadku niespełnienia kryterium porządkowego zbioru. Operację tę wykonujemy dotąd, aż cały zbiór zostanie posortowany.
Algorytm sortowania bąbelkowego przy porządkowaniu zbioru nieposortowanego ma klasę czasowej złożoności obliczeniowej równą Θ(n2). Sortowanie odbywa się w miejscu.
*Przykład :*
Jako przykład działania algorytmu sortowania bąbelkowego posortujemy przy jego pomocy 5-cio elementowy zbiór liczb {5 4 3 2 1}, który wstępnie jest posortowany w kierunku odwrotnym, co możemy uznać za przypadek najbardziej niekorzystny, ponieważ wymaga przestawienia wszystkich elementów.

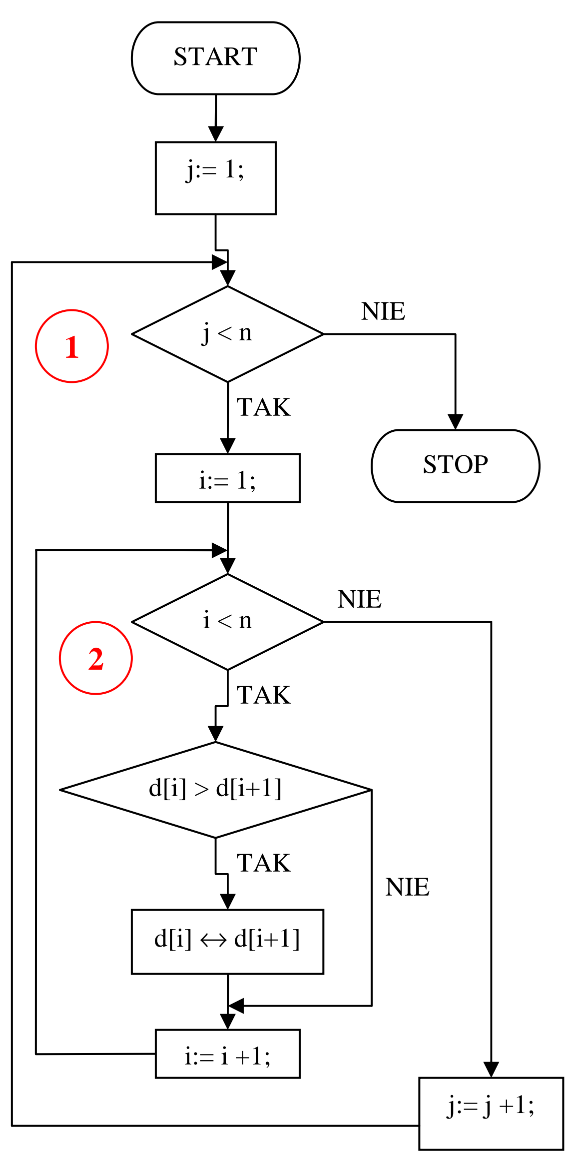
| Obieg | Zbiór | Opis operacji |
|---|---|---|
| 1 | 5 4 3 2 1 | Rozpoczynamy od pierwszej pary, która wymaga wymiany elementów |
| | 4 5 3 2 1 | Druga para też wymaga zamiany elementów |
| | 4 3 5 2 1 | Wymagana wymiana elementów |
| | 4 3 2 5 1 | Ostatnia para również wymaga wymiany elementów |
| | 4 3 2 1 5 | Stan po pierwszym obiegu. Zwróć uwagę, iż najstarszy element (5) znalazł się na końcu zbioru, a najmłodszy (1) przesunął się o jedną pozycję w lewo. |
| 2 | 4 3 2 1 5 | Para wymaga wymiany |
| | 3 4 2 1 5 | Para wymaga wymiany |
| | 3 2 4 1 5 | Para wymaga wymiany |
| | 3 2 1 4 5 | Elementy są w dobrej kolejności, zamiana nie jest konieczna. |
| | 3 2 1 4 5 | Stan po drugim obiegu. Zwróć uwagę, iż najmniejszy element (1) znów przesunął się o jedną pozycję w lewo. Z obserwacji tych można wywnioskować, iż po każdym obiegu najmniejszy element wędruje o jedną pozycję ku początkowi zbioru. Najstarszy element zajmuje natomiast swe miejsce końcowe. |
| 3 | 3 2 1 4 5 | Para wymaga wymiany |
| | 2 3 1 4 5 | Para wymaga wymiany |
| | 2 1 3 4 5 | Dobra kolejność |
| | 2 1 3 4 5 | Dobra kolejność |
| | 2 1 3 4 5 | Stan po trzecim obiegu. Wnioski te same. |
| 4 | 2 1 3 4 5 | Para wymaga wymiany |
| | 1 2 3 4 5 | Dobra kolejność |

|  | 1 2 3 4 5 | Dobra kolejność |
|---|---|---|
|  | 1 2 3 4 5 | Dobra kolejność |
|  | 1 2 3 4 5 | Zbiór jest posortowany. Koniec |

Posortowanie naszego zbioru wymaga 4 obiegów. Jest to oczywiste: w przypadku najbardziej niekorzystnym najmniejszy element znajduje się na samym końcu zbioru wejściowego. Każdy obieg przesuwa go o jedną pozycję w kierunku początku zbioru. Takich przesunięć należy wykonać n - 1 (n - ilość elementów w zbiorze).

Algorytm sortowania bąbelkowego, w przeciwieństwie do algorytmu sortowania naiwnego, nie przerywa porównywania par elementów po napotkaniu pary nie spełniającej założonego porządku. Po zamianie kolejności elementów sprawdzana jest kolejna para elementów sortowanego zbioru. Dzięki temu podejściu rośnie efektywność algorytmu oraz zmienia się klasa czasowej złożoności obliczeniowej z $\Theta(n^3)$ na $\Theta(n^2)$.

***Schemat blokowy :***




Sortowanie wykonywane jest w dwóch zagnieżdżonych pętlach. Pętla zewnętrzna nr 1 kontrolowana jest przez zmienną j. Wykonuje się ona n - 1 razy. Wewnątrz pętli nr 1 umieszczona jest pętla nr 2 sterowana przez zmienną i. Wykonuje się ona również n - 1 razy. W efekcie algorytm wykonuje w sumie: $T_1(n) = (n - 1)^2 = n^2 - 2n + 1$

obiegów pętli wewnętrznej, po których zakończeniu zbiór zostanie posortowany.

Sortowanie odbywa się wewnątrz pętli nr 2. Kolejno porównywany jest i-ty element z elementem następnym. Jeśli elementy te są w złej kolejności, to zostają zamienione miejscami. W tym miejscu jest najważniejsza różnica pomiędzy algorytmem sortowania bąbelkowego a algorytmem sortowania naiwnego. Ten drugi w momencie napotkania elementów o złej kolejności zamienia je miejscami i rozpoczyna cały proces sortowania od początku. Algorytm sortowania bąbelkowego wymienia miejscami źle ułożone elementy sortowanego zbioru i przechodzi do następnej pary zwiększając indeks i o 1. Dzięki takiemu podejściu rośnie efektywność, co odzwierciedla klasa czasowej złożoności obliczeniowej:
Sortowanie naiwne - $\Theta(n^3)$;

Sortowanie bąbelkowe - $\Theta(n^2)$

| Cechy Algorytmu Sortowania Bąbelkowego wersja nr 1 | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta(n^2)$ |
| klasa złożoności obliczeniowej typowa | $\Theta(n^2)$ |
| klasa złożoności obliczeniowej pesymistyczna | $\Theta(n^2)$ |
| Sortowanie w miejscu | TAK |
| Stabilność | TAK |

Podany wyżej algorytm sortowania bąbelkowego można zoptymalizować pod względem czasu wykonania. Jeśli przyjrzymy się dokładnie obiegom wykonywanym w tym algorytmie, to zauważymy bardzo istotną rzecz : Po wykonaniu pełnego obiegu w algorytmie sortowania bąbelkowego najstarszy element wyznaczony przez przyjęty porządek zostaje umieszczony na swoim właściwym miejscu - na końcu zbioru.

Wniosek ten jest oczywisty. W każdej kolejnej parze porównywanych elementów element starszy przechodzi na drugą pozycję. W kolejnej parze jest on na pierwszej pozycji, a skoro jest najstarszym, to po porównaniu znów przejdzie na pozycję drugą itd. - jest jakby ciągnięty na koniec zbioru (jak bąbelek powietrza wypływający na powierzchnię wody).

Spójrzmy na poniższy przykład.

Wykonamy jeden obieg sortujący dla zbioru pięcioelementowego *{ **9** 3 1 7 0 }*. Elementem najstarszym jest pierwszy element - liczba 9.

| Obieg | Zbiór | Opis operacji |
|---|---|---|
| 1 | **9 3** 1 7 0 | Para wymaga przestawienia elementów. Element najstarszy przejdzie na drugą pozycję w parze. |
| | 3 **9 1** 7 0 | Konieczne przestawienie elementów. Element najstarszy znów trafi na pozycję drugą w parze. |
| | 3 1 **9 7** 0 | Konieczne przestawienie elementów. |
| | 3 1 7 **9 0** | Ostatnia para również wymaga przestawienia elementów. |
| | 3 1 7 0 **9** | Koniec obiegu. Najstarszy element znalazł się na końcu zbioru. |

Zwróćmy uwagę, że po każdym obiegu na końcu zbioru tworzy się podzbiór uporządkowanych najstarszych elementów. Zatem w kolejnych obiegach możemy pomijać sprawdzanie ostatnich elementów - liczebność zbioru do posortowania z każdym obiegiem maleje o 1.

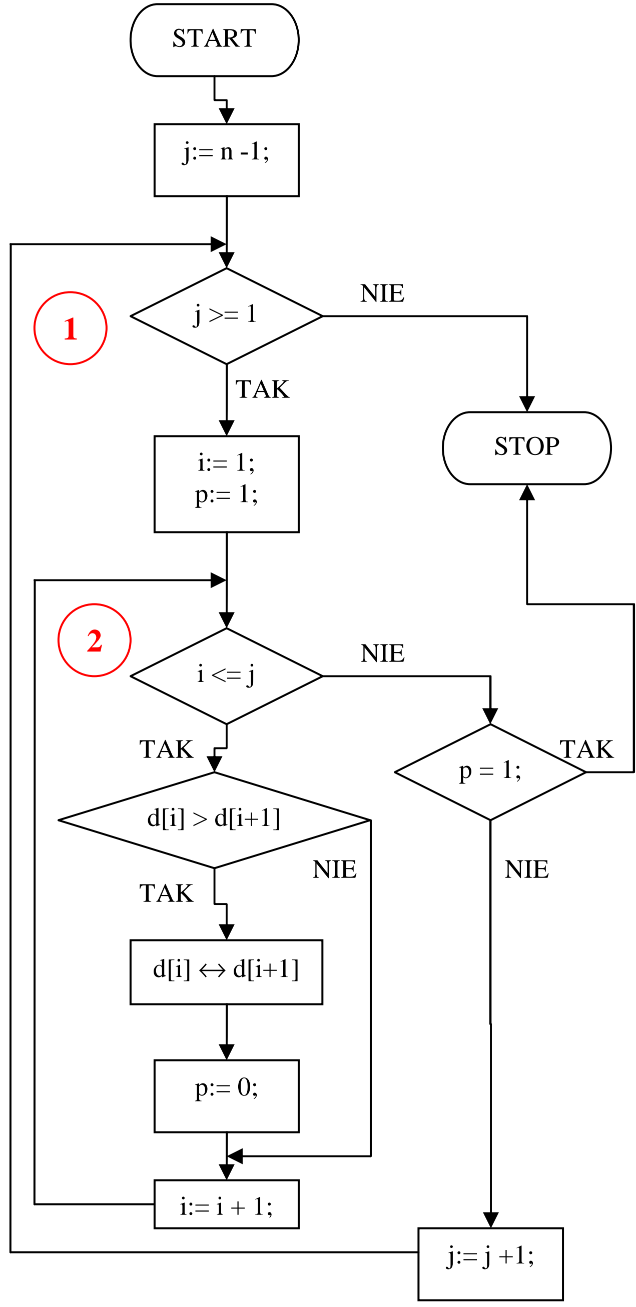
Zwróćmy uwagę, że pozbyliśmy się tutaj operacji „pustych” to znaczy takich, które nic nie robią nam w zbiorze. Pozbycie się tych operacji nastąpiło oczywiście w petli zewnętrznej (pętla 1).

Możliwa jest dalsza redukcja operacji pustych, jeśli będziemy sprawdzać, czy w pętli wewnętrznej były przestawiane elementy (czyli czy wykonano operacje sortujące). Jeśli nie, to zbiór jest już posortowany i możemy zakończyć pracę algorytmu.

***Przykład :***
Posortujmy zbiór { 3 1 0 7 9 } zgodnie z wprowadzoną modyfikacją (sprawdzeniem, czy w petli wewnętrznej były przestawiane elementy).

| Obieg | Zbiór | Opis operacji |
|---|---|---|
| 1 | 3 1 0 7 9 | Konieczne przestawienie elementów |
| | 1 3 0 7 9 | Konieczne przestawienie elementów |
| | 1 0 3 7 9 | Elementy w dobrej kolejności |
| | 1 0 3 7 9 | Elementy w dobrej kolejności |
| | 1 0 3 7 9 | Koniec pierwszego obiegu. Ponieważ były przestawienia elementów, sortowanie kontynuujemy |
| 2 | 1 0 3 7 9 | Konieczne przestawienie elementów |
| | 0 1 3 7 9 | Elementy w dobrej kolejności |
| | 0 1 3 7 9 | Elementy w dobrej kolejności |
| | 0 1 3 7 9 | Koniec drugiego obiegu. Było przestawienie elementów, zatem sortowanie kontynuujemy |
| 3 | 0 1 3 7 9 | Elementy w dobrej kolejności |
| | 0 1 3 7 9 | Elementy w dobrej kolejności |
| | 0 1 3 7 9 | Koniec trzeciego obiegu. Nie było przestawień elementów, kończymy sortowanie. Wykonaliśmy o 1 obieg sortujący mniej. |

***Schemat blokowy uwzględniający wszystkie powyższe modyfikacje*** :




Pętla zewnętrzna nr 1 zlicza obiegi wstecz, tzn. pierwszy obieg ma numer n-1. Dzięki takiemu podejściu w zmiennej j mamy zawsze numer ostatniego elementu, do którego ma dojść pętla wewnętrzna nr 2. Ta zmiana wymaga również odwrotnej iteracji zmiennej j.
Pętla wewnętrzna (2) sprawdza w warunku kontynuacji, czy wykonała j obiegów, a nie jak poprzednio n-1 obiegów. Dzięki temu po każdym obiegu pętli nr 1 (zewnętrznej) pętla nr 2 będzie wykonywać o jeden obieg mniej.
Dodatkowo, przed wejściem do pętli sortującej nr 2 ustawiamy zmienną pomocniczą p. Jeśli w pętli zajdzie potrzeba przestawienia elementów, to zmienna p jest zerowana. Po wykonaniu pętli sortującej sprawdzamy, czy zmienna p jest ustawiona. Jeśli tak, to przestawienie elementów nie wystąpiło, zatem kończymy algorytm. W przeciwnym razie wykonujemy kolejny obieg pętli nr 1.
Pozostała część algorytmu nie jest zmieniona - w pętli wewnętrznej nr 2 sprawdzamy, czy element d[i] jest w złej kolejności z elementem d[i+1]. Sprawdzany warunek spowoduje posortowanie zbioru rosnąco. Przy sortowaniu malejącym zmieniamy relację większości na relację mniejszości. Jeśli warunek jest spełniony, zamieniamy miejscami element d[i] z elementem d[i+1], po czym kontynuujemy pętlę nr 2 zwiększając o 1 indeks i. Po każdym zakończeniu pętli nr 2 indeks j jest zmniejszany o 1.
Zbiór będzie posortowany, jeśli po wykonaniu wewnętrznego obiegu sortującego nie wystąpi ani jedno przestawienie elementów porządkowanego zbioru.

Czy algorytm sortowania bąbelkowego można jeszcze ulepszyć? Tak, ale zaczynamy już osiągać kres jego możliwości, ponieważ ulepszenia polegają jedynie na redukcji operacji pustych. Wykorzystamy informację o miejscu wystąpienia zamiany elementów (czyli o miejscu wykonania operacji sortującej).

Jeśli w obiegu sortującym wystąpi pierwsza zamiana na pozycji i-tej, to w kolejnym obiegu będziemy rozpoczynali sortowanie od pozycji o jeden mniejszej (chyba, że pozycja *i*-ta była

pierwszą pozycją w zbiorze). Dlaczego? Odpowiedź jest prosta. Zamiana spowodowała, iż młodszy element znalazł się na pozycji i-tej. Ponieważ w obiegu sortującym młodszy element zawsze przesuwa się o 1 pozycję w kierunku początku zbioru, to nie ma sensu sprawdzanie pozycji od 1 do i-2, ponieważ w poprzednim obiegu zostały one już sprawdzone, nie wystąpiła na nich zamiana elementów, zatem elementy na pozycjach od 1 do i-2 są chwilowo w dobrej kolejności względem siebie. Nie mamy tylko pewności co do pozycji i-1-szej oraz i-tej, ponieważ ostatnia zamiana elementów umieściła na i-tej pozycji młodszy element, który być może należy wymienić z elementem na pozycji wcześniejszej, czyli i-1. W ten sposób określimy początkową pozycję, od której rozpoczniemy sortowanie elementów w następnym obiegu sortującym.

Ostatnia zamiana elementów wyznaczy pozycję końcową dla następnego obiegu. Wiemy, iż w każdym obiegu sortującym najstarszy element jest zawsze umieszczany na swojej docelowej pozycji. Jeśli ostatnia zamiana elementów wystąpiła na pozycji i-tej, to w następnym obiegu porównywanie elementów zakończymy na pozycji o 1 mniejszej - w ten sposób nie będziemy sprawdzać już najstarszego elementu z poprzedniego obiegu.

Sortowanie prowadzimy dotąd, aż w obiegu sortującym nie wystąpi ani jedna zamiana elementów.

Teoretycznie powinno to zoptymalizować algorytm, ponieważ są sortowane tylko niezbędne fragmenty zbioru - pomijamy obszary posortowane, które tworzą się na końcu i na początku zbioru. Oczywiście zysk nie będzie oszałamiający w przypadku zbioru nieuporządkowanego lub posortowanego odwrotnie (może się zdarzyć, iż ewentualne korzyści czasowe będą mniejsze od czasu wykonywania dodatkowych operacji). Jednakże dla zbiorów w dużym stopniu uporządkowanych możemy uzyskać całkiem rozsądny algorytm sortujący prawie w czasie liniowym $\Theta(n)$.

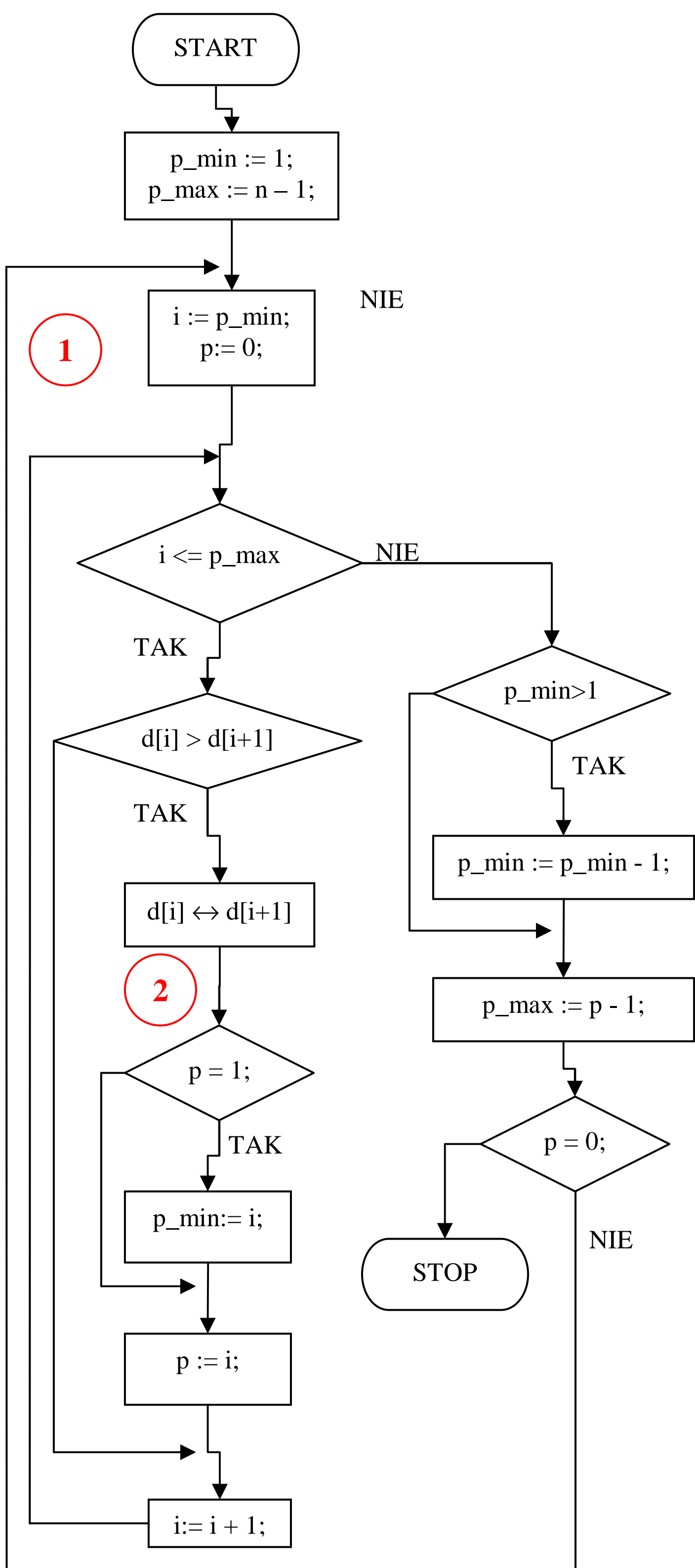
*Zmienne pomocnicze :*

i - zmienna sterująca pętli, $i \in N$
p_min - dolna granica pozycji sortowanych elementów, $p_min \in N$
p_max - górna granica pozycji sortowanych elementów, $p_max \in N$
p - numer pozycji zamiany elementów, $p \in N$

***Lista kroków :***

Krok 1 : p_min := 1 ; p_max := n-1;
Krok 2 : p := 0;
Krok 3 : Dla i:= p_min ...... p_max wykonuj kroki 4...7
Krok 4 : Jeśli d[i] <= d[i + 1], to wykonaj następny obieg pętli z kroku 3
Krok 5 : Zamień wartości tablicy d[i] z d[i+1] ( d[i] $\leftrightarrow$ d[i+1] )
Krok 6 : Jeśli p = 0, to p_min := i;
Krok 7 : p := i;
Krok 8 : Jeśli p_min > 1, to p_min := p_min – 1 ;
Krok 9 : p_max := p – 1;
Krok 10: Jeśli p > 0, to wróć do kroku 2
Krok 11: Zakończ algorytm

***Schemat blokowy uwzględniający wszystkie powyższe modyfikacje*** :




Zmienna p_min przechowuje numer pozycji, od której rozpoczyna się sortowanie zbioru. W pierwszym obiegu sortującym rozpoczynamy od pozycji nr 1. Zmienna p_max przechowuje numer ostatniej pozycji do sortowania. Pierwszy obieg sortujący kończymy na pozycji n-1, czyli na przedostatniej.

Pętla numer 1 wykonywana jest dotąd, aż w wewnętrznej pętli nr 2 nie wystąpi żadna zamiana elementów. Zmienna p pełni w tej wersji algorytmu nieco inną rolę niż poprzednio. Mianowicie będzie przechowywała numer pozycji, na której algorytm ostatnio dokonał wymiany elementów. Na początku wpisujemy do p wartość 0, która nie oznacza żadnej pozycji w zbiorze. Zatem jeśli ta wartość zostanie zachowana, uzyskamy pewność, iż zbiór jest posortowany, ponieważ nie dokonano wymiany elementów.

Wewnętrzną pętlę sortującą rozpoczynamy od pozycji pmin. W pętli sprawdzamy kolejność elementu i-tego z elementem następnym. Jeśli kolejność jest zła, wymieniamy miejscami te dwa elementy. Po wymianie sprawdzamy, czy jest to pierwsza wymiana - zmienna p ma wtedy wartość 0. Jeśli tak, to numer pozycji, na której dokonano wymiany umieszczamy w pmin. Numer ten zapamiętujemy również w zmiennej p. Zwróć uwagę, iż dzięki takiemu podejściu p zawsze będzie przechowywało numer pozycji ostatniej wymiany - jest to zasada zwana "ostatni zwycięża".

Po sprawdzeniu elementów przechodzimy do następnej pozycji zwiększając i o 1 i kontynuujemy pętlę, aż do przekroczenia pozycji p_max. Wtedy pętla wewnętrzna zakończy się. Jeśli w pętli nr 2 była dokonana zamiana elementów, to pmin zawiera numer pozycji pierwszej zamiany. Jeśli nie jest to pierwsza pozycja w zbiorze, pmin zmniejszamy o 1, aby pętla

sortująca rozpoczynała od pozycji poprzedniej w stosunku do pozycji pierwszej zamiany elementów.
Pozycję ostatnią zawsze ustalamy o 1 mniejszą od numeru pozycji końcowej zamiany elementów.
Na koniec sprawdzamy, czy faktycznie doszło do zamiany elementów. Jeśli tak, to p jest większe od 0, gdyż zawiera numer pozycji w zbiorze, na której algorytm wymienił miejscami elementy. W takim przypadku pętlę nr 1 rozpoczynamy od początku. W przeciwnym razie kończymy, zbiór jest uporządkowany.

| **Cechy Algorytmu Sortowania Bąbelkowego wersja nr 2** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta$ (n) |
| klasa złożoności obliczeniowej typowa | $\Theta$ ($n^2$) |
| klasa złożoności obliczeniowej pesymistyczna | $\Theta$ ($n^2$) |
| Sortowanie w miejscu | TAK |
| Stabilność | TAK |

## *8.4. Sortowanie przez wybór.*

Idea algorytmu sortowania przez wybór jest bardzo prosta. Załóżmy, iż chcemy posortować zbiór liczbowy rosnąco. Zatem element najmniejszy powinien znaleźć się na pierwszej pozycji. Szukamy w zbiorze elementu najmniejszego i wymieniamy go z elementem na pierwszej pozycji. W ten sposób element najmniejszy znajdzie się na swojej docelowej pozycji.
W identyczny sposób postępujemy z resztą elementów należących do zbioru. Znów wyszukujemy element najmniejszy i zamieniamy go z elementem na drugiej pozycji. Otrzymamy dwa posortowane elementy. Procedurę kontynuujemy dla pozostałych elementów dotąd, aż wszystkie będą posortowane.
Algorytm sortowania przez wybór posiada klasę czasowej złożoności obliczeniowej równą $\Theta(n^2)$. Sortowanie odbywa się w miejscu.

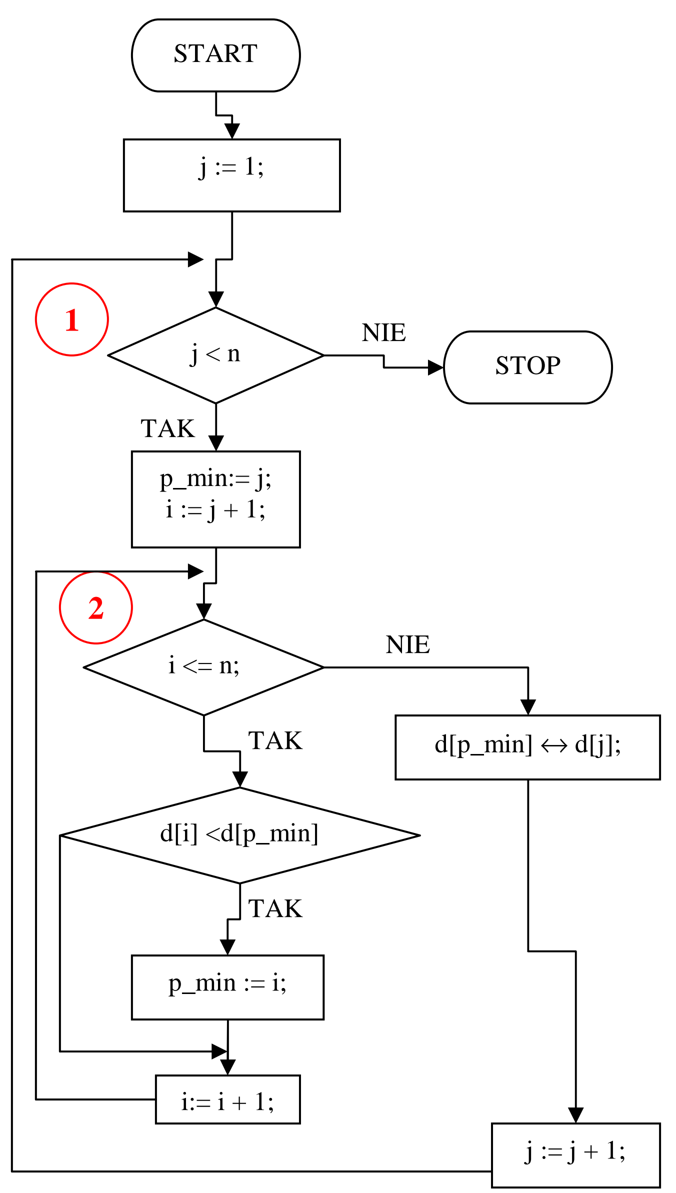
Przykład : posortujmy tą metodą zbiór {4 7 2 9 3}. Kolorem zielonym oznaczono elementy zbioru, które są już posortowane.

| **Zbiór** | **Opis operacji** |
|---|---|
| 4 7 2 9 3 | Wyszukujemy najmniejszy element w zbiorze. Jest nim liczba 2. |
| 2 7 4 9 3 | Znaleziony element minimalny wymieniamy z pierwszym elementem zbioru - liczbą 4 |
| 2 7 4 9 3 | Wśród pozostałych elementów wyszukujemy element najmniejszy. Jest nim liczba 3. |
| 2 3 4 9 7 | Znaleziony element minimalny wymieniamy z drugim elementem zbioru - liczbą 7. |
| 2 3 4 9 7 | Znajdujemy kolejny element minimalny - liczbę 4. |
| 2 3 4 9 7 | Wymieniamy go z samym sobą - element ten nie zmienia zatem swojej pozycji w zbiorze. |
| 2 3 4 9 7 | Znajdujemy kolejny element minimalny |

| 2 3 4 7 9 | Wymieniamy go z liczbą 9 |
|---|---|
| 2 3 4 7 9 | Ostatni element jest zawsze na właściwej pozycji. Sortowanie zakończone |

Podana metoda sortuje zbiór rosnąco. Jeśli chcemy posortować zbiór malejąco, to zamiast elementu minimalnego poszukujemy elementu maksymalnego. Pozostała część procedury sortującej nie ulega zmianie.

***Schemat blokowy :***




Pętla zewnętrzna (1) sterowana zmienną j wyznacza kolejne elementy zbioru o indeksach od 1 do n - 1, w których zostaną umieszczone elementy minimalne. Na początku tej pętli zakładamy, iż elementem minimalnym jest element d[j] i zapamiętujemy jego indeks w zmiennej $p_{min}$.

W pętli numer 2 sterowanej zmienną i porównujemy pozostałe elementy zbioru z elementem d[$p_{min}$]. Jeśli element zbioru d[i] jest mniejszy od elementu d[$p_{min}$], to znaleźliśmy nowy element minimalny. W takim przypadku zapamiętujemy jego pozycję w $p_{min}$ i kontynuujemy pętlę wewnętrzną.

Po zakończeniu pętli wewnętrznej $p_{min}$ zawiera indeks elementu minimalnego. Zamieniamy miejscami element d[j] z elementem d[$p_{min}$]. Dzięki tej operacji element minimalny znajduje się na swojej docelowej pozycji. Zwiększamy j przechodząc do kolejnego elementu zbioru i kontynuujemy pętlę zewnętrzną.

| Cechy Algorytmu Sortowania Przez Wybór | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta(n^2)$ |
| klasa złożoności obliczeniowej typowa | $\Theta(n^2)$ |
| klasa złożoności obliczeniowej pesymistyczna | $\Theta(n^2)$ |
| Sortowanie w miejscu | TAK |
| Stabilność | NIE |

## *8.5. Sortowanie przez wstawianie.*

Algorytm sortowania przez wstawianie można porównać do sposobu układania kart pobieranych z talii. Najpierw bierzemy pierwszą kartę. Następnie pobieramy kolejne, aż do wyczerpania talii. Każdą pobraną kartę porównujemy z kartami, które już trzymamy w ręce i szukamy dla niej miejsca przed pierwszą kartą starszą (młodszą w przypadku porządku malejącego). Gdy znajdziemy takie miejsce, rozsuwamy karty i nową wstawiamy na przygotowane w ten sposób miejsce (stąd pochodzi nazwa algorytmu - sortowanie przez wstawianie, ang. Insertion Sort). Jeśli nasza karta jest najstarsza (najmłodsza), to umieszczamy ją na samym końcu. Tak porządkujemy karty. A jak przenieść tę ideę do świata komputerów i zbiorów liczbowych?

Algorytm sortowania przez wstawianie będzie składał się z dwóch pętli. Pętla główna (zewnętrzna) symuluje pobieranie kart, czyli w tym wypadku elementów zbioru. Odpowiednikiem kart na ręce jest tzw. lista uporządkowana (ang. sorted list), którą sukcesywnie będziemy tworzyli na końcu zbioru (istnieje też odmiana algorytmu umieszczająca listę uporządkowaną na początku zbioru). Pętla sortująca (wewnętrzna) szuka dla pobranego elementu miejsca na liście uporządkowanej. Jeśli takie miejsce zostanie znalezione, to elementy listy są odpowiednio rozsuwane, aby tworzyć miejsce na nowy element i element wybrany przez pętlę główną trafia tam. W ten sposób lista uporządkowana rozrasta się. Jeśli na liście uporządkowanej nie ma elementu większego od wybranego, to element ten trafia na koniec listy. Sortowanie zakończymy, gdy pętla główna wybierze wszystkie elementy zbioru.

Algorytm sortowania przez wstawianie posiada klasę czasowej złożoności obliczeniowej równą $\Theta(n^2)$. Sortowanie odbywa się w miejscu.

Najważniejszą operacją w opisywanym algorytmie sortowania jest wstawianie wybranego elementu na listę uporządkowaną. Zasady są następujące:

1. Na początku sortowania lista uporządkowana zawiera tylko jeden, ostatni element zbioru. Jednoelementowa lista jest zawsze uporządkowana.

2. Ze zbioru zawsze wybieramy element leżący tuż przed listą uporządkowaną. Element ten zapamiętujemy w zewnętrznej zmiennej. Miejsce, które zajmował, możemy potraktować jak puste.

3. Wybrany element porównujemy z kolejnymi elementami listy uporządkowanej.

4. Jeśli natrafimy na koniec listy, element wybrany wstawiamy na puste miejsce - lista rozrasta się o nowy element.

5. Jeśli element listy jest większy od wybranego, to element wybrany wstawiamy na puste miejsce - lista rozrasta się o nowy element.
6. Jeśli element listy nie jest większy od wybranego, to element listy przesuwamy na puste miejsce. Dzięki tej operacji puste miejsce wędruje na liście przed kolejny element. Kontynuujemy porównywanie, aż wystąpi sytuacja z punktu 4 lub 5.

***Przykład :***

Posortujmy zbiór { 7 3 8 5 2 }.

| Zbiór | Opis operacji |
|---|---|
| 7 3 8 5 2 | Ostatni element jest zalążkiem listy uporządkowanej. |
| 5<br>7 3 8 2 | Ze zbioru wybieramy element leżący tuż przed listą uporządkowaną. |
| 5<br>7 3 8 2 | Wybrany element porównujemy z elementem listy. |
| 5<br>7 3 8 2 <<< | Ponieważ element listy jest mniejszy od elementu wybranego, to przesuwamy go na puste miejsce. |
| 7 3 8 2 5 | Na liście nie ma już więcej elementów do porównania, więc element wybrany wstawiamy na puste miejsce. Lista uporządkowana zawiera już dwa elementy. |
| 8<br>7 3 2 5 | Ze zbioru wybieramy 8 |
| 8<br>7 3 2 5 | 8 porównujemy z 2 |
| 8<br>7 3 2 <<< 5 | 2 jest mniejsze, zatem przesuwamy je na puste miejsce. |
| 8<br>7 3 2 5 | 8 porównujemy z 5 |
| 8<br>7 3 2 5 <<< | 5 jest mniejsze, przesuwamy je na puste miejsce |
| 7 3 2 5 8 | Lista nie zawiera więcej elementów, zatem 8 wstawiamy na puste miejsce. Na liście uporządkowanej mamy już trzy elementy. |
| 3<br>7 2 5 8 | Ze zbioru wybieramy 3 |
| 3<br>7 2 5 8 | 3 porównujemy z 2 |
| 3<br>7 2 <<< 5 8 | 2 jest mniejsze, wędruje zatem na puste miejsce |
| 3<br>7 2 5 8 | 3 porównujemy z 5. |
| 7 2 3 5 8 | Tym razem mniejszy jest element wybrany. Znaleźliśmy jego miejsce na liście, więc wstawiamy go na puste miejsce. Lista zawiera już 4 elementy. |
| 7<br>2 3 5 8 | Ze zbioru wybieramy ostatni element - liczbę 7. |

| | |
|---|---|
| 7<br>2 3 5 8 | 7 porównujemy z 2 |
| 7<br>2 <<< 3 5 8 | 2 jest mniejsze, przesuwamy je na puste miejsce |
| 7<br>2 3 5 8 | 7 porównujemy z 3 |
| 7<br>2 3 <<< 5 8 | 3 jest mniejsze, przesuwamy je na puste miejsce |
| 7<br>2 3 5 8 | 7 porównujemy z 5 |
| 7<br>2 3 5 <<< 8 | 5 jest mniejsze, przesuwamy je na puste miejsce |
| 7<br>2 3 5 8 | 7 porównujemy z 8 |
| 2 3 5 7 8 | Element wybrany jest mniejszy, wstawiamy go na puste miejsce. Lista uporządkowana objęła już cały zbiór. Sortowanie jest zakończone. |

| **Cechy Algorytmu Sortowania Przez Wstawianie** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta(n)$ |
| klasa złożoności obliczeniowej typowa | $\Theta(n^2)$ |
| klasa złożoności obliczeniowej pesymistyczna | $\Theta(n^2)$ |
| Sortowanie w miejscu | TAK |
| Stabilność | TAK |

***Specyfikacja algorytmu :***

*Dane wejściowe :*

n - liczba elementów w sortowanym zbiorze, $n \in N$
d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do n.

*Dane wyjściowe :*

d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do n

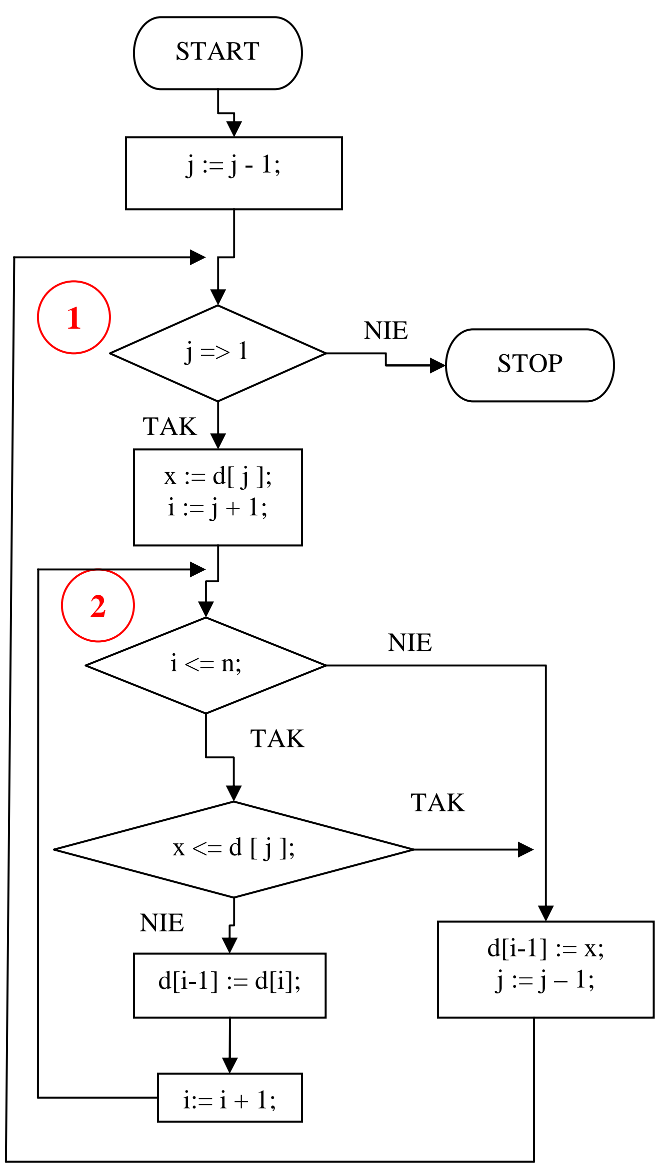
*Zmienne pomocnicze :*

i, j - zmienne sterujące pętli, $i, j \in N$
x - zawiera wybrany ze zbioru element

***Lista kroków :***

Krok 1 : Dla j = n - 1, n - 2, ..., 1: wykonuj kroki 2...4
Krok 2 : x := d[j]; i := j + 1;
Krok 3 : Dopóki ( i <= n ) i ( x > d[i] ) wykonuj d[i - 1] := d[i]; i := i + 1
Krok 4 : d[i - 1] := x
Krok 5 : Zakończ

:
***Schemat blokowy***




Pętlę główną rozpoczynamy od przedostatniej pozycji w zbiorze. Element na ostatniej pozycji jest zalążkiem listy uporządkowanej. Dlatego licznik pętli nr 1 przyjmuje wartość początkową j = n - 1.

Ze zbioru wybieramy element d[j] i umieszczamy go w zmiennej pomocniczej x. Miejsce zajmowane przez ten element staje się puste.
Pętlę wewnętrzną rozpoczynamy od pozycji następnej w stosunku do j. Pozycja ta zawiera pierwszy element listy uporządkowanej, która tworzona jest na końcu sortowanego zbioru. Pętlę wewnętrzną przerywamy w dwóch przypadkach - gdy licznik pętli wyjdzie poza indeks ostatniego elementu w zbiorze lub gdy element wybrany, przechowywany w zmiennej pomocniczej x, jest mniejszy lub równy bieżąco testowanemu elementowi listy uporządkowanej (dla sortowania malejącego należy zastosować w tym miejscu relację większy lub równy). W obu przypadkach na puste miejsce ma trafić zawartość zmiennej x i pętla zewnętrzna jest kontynuowana. Zauważ, iż pozycja tego miejsca w zbiorze jest równa i - 1.

Jeśli żaden z warunków przerwania pętli wewnętrznej nr 2 nie wystąpi, to przesuwamy bieżący element listy na puste miejsce i kontynuujemy pętlę wewnętrzną.

Podsumowując: pętla zewnętrzna wybiera ze zbioru kolejne elementy o indeksach od n - 1 do 1, pętla wewnętrzna szuka dla wybranych elementów miejsca wstawienia na liście uporządkowanej, po znalezieniu którego pętla zewnętrzna wstawia wybrany element na listę. Gdy pętla zewnętrzna przebiegnie wszystkie elementy o indeksach od n - 1 do 1, zbiór będzie posortowany.

## 8.6. Sortowanie metodą Shella.

W latach 50-tych XX wieku informatyk Donald Shell zauważył, iż algorytm sortowania przez wstawianie pracuje bardzo efektywnie w przypadku gdy zbiór jest w dużym stopniu uporządkowany, a gdy zbiór jest nieuporządkowany to algorytm ten pracuje nieefektywnie. Dzieje się to dlatego, iż elementy są przesuwane w każdym obiegu o jedną pozycję przy wstawianiu elementu wybranego na listę uporządkowaną.

Pomysł Shella polegał na tym, iż sortowany zbiór dzielimy na podzbiory, których elementy są odległe od siebie w sortowanym zbiorze o pewien odstęp *h*. Każdy z tych podzbiorów sortujemy algorytmem przez wstawianie. Następnie odstęp zmniejszamy. Powoduje to powstanie nowych podzbiorów (będzie ich już mniej). Sortowanie powtarzamy i znów zmniejszamy odstęp, aż osiągnie on wartość 1. Wtedy sortujemy już normalnie za pomocą Instertion Sort. Jednakże z uwagi na wcześniejsze obiegi sortujące mamy ułatwione zadanie, ponieważ zbiór został w dużym stopniu uporządkowany. Dzięki początkowym dużym odstępom elementy były przesuwane w zbiorze bardziej efektywnie - na duże odległości. W wyniku otrzymujemy najlepszy pod względem szybkości czasu wykonania algorytm sortujący w klasie $\Theta(n^2)$. Algorytm ten nosi również nazwę algorytmu sortowania przez wstawianie z malejącym odstępem (ang. Diminishing Increment Sort).

Efektywność algorytmu sortowania metodą Shella zależy w dużym stopniu od ciągu przyjętych odstępów. Pierwotnie Shell proponował pierwszy odstęp równy połowie liczby elementów w sortowanym zbiorze. Kolejne odstępy otrzymujemy dzieląc odstęp przez 2 (dzielenie całkowitoliczbowe).

***Przykład :***
Posortujemy metodą Shella zbiór ośmiu liczb: { 4 2 9 5 6 3 8 1 } w porządku rosnącym. Zbiór posiada osiem elementów, zatem przyjmiemy na wstępie odstęp *h* równy 4. Taki odstęp podzieli zbiór na 4 podzbiory, których elementy będą elementami zbioru wejściowego odległymi od siebie o 4 pozycje. Każdy z otrzymanych podzbiorów sortujemy algorytmem sortowania przez wstawianie. Ponieważ zbiory te są dwuelementowe, to sortowanie pędzie polegało na porównaniu pierwszego elementu podzbioru z elementem drugim i ewentualną zamianę ich miejsc, jeśli będą w niewłaściwym porządku.

| | Podział, h=4 | | | | | | | | Sortowanie | | Wynik | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 2 | 9 | 5 | 6 | 3 | 8 | 1 | - Zbiór wejściowy | | | | | | | | | |
| 1 | 4 | | | | 6 | | | | 4 | 6 | 4 | | | | 6 | | | |
| 2 | | 2 | | | | 3 | | | 2 | 3 | | 2 | | | | 3 | | |
| 3 | | | 9 | | | | 8 | | 8 | 9 | | | 8 | | | | 9 | |
| 4 | | | | 5 | | | | 1 | 1 | 5 | | | | 1 | | | | 5 |
| | | | | | | | | | | Zbiór wyjściowy - | 4 | 2 | 8 | 1 | 6 | 3 | 9 | 5 |

Zmniejszamy odstęp h o połowę, więc h = 2. Zbiór podstawowy zostanie podzielony na dwa podzbiory. Każdy z tych podzbiorów sortujemy przez wstawianie:

| | Podział, h=2 | | | | | | | | Sortowanie | | | | Wynik | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 2 | 8 | 1 | 6 | 3 | 9 | 5 | - Zbiór wejściowy | | | | | | | | | | | |
| 1 | 4 | | 8 | | 6 | | 9 | | 4 | 6 | 8 | 9 | 4 | | 6 | | 8 | | 9 | |
| 2 | | 2 | | 1 | | 3 | | 5 | 1 | 2 | 3 | 5 | | 1 | | 2 | | 3 | | 5 |
| | | | | | | | | | | | | Zbiór wyjściowy - | 4 | 1 | 6 | 2 | 8 | 3 | 9 | 5 |

Zmniejszamy odstęp h o połowę, h = 1. Taki odstęp nie dzieli zbioru wejściowego na podzbiory, więc teraz będzie sortowany przez wstawianie cały zbiór. Jednak algorytm sortujący ma ułatwioną pracę, ponieważ dzięki poprzednim dwóm obiegom zbiór został częściowo uporządkowany - elementy małe zbliżyły się do początku zbioru, a elementy duże do końca.

| | Podział, h=1 | Sortowanie | Wynik |
|---|---|---|---|
| | 4 1 6 2 8 3 9 5 | - Zbiór wejściowy | |
| 1 | 4 1 6 2 8 3 9 5 | 1 2 3 4 5 6 8 9 | 1 2 3 4 5 6 8 9 |
| | | Zbiór wyjściowy - | 1 2 3 4 5 6 8 9 |

***Dobór optymalnych odstępów.***

Kluczowym elementem wpływającym na efektywność sortowania metodą Shella jest właściwy dobór ciągu odstępów. Okazuje się, iż ciąg zaproponowany przez twórcę algorytmu jest jednym z najgorszych, ponieważ w kolejnych podzbiorach uczestniczą wielokrotnie te same elementy.
Problem optymalnych odstępów w algorytmie sortowania metodą Shella nie został rozwiązany matematycznie, ponieważ w ogólnym przypadku jest niezwykle trudny. Wielu badaczy proponowało na wybór tych odstępów różne ciągi liczbowe otrzymując lepsze lub gorsze rezultaty.
Na przykład profesor Donald Knuth zbadał bardzo dokładnie algorytm sortowania metodą Shella i doszedł do wniosku, iż dobry ciąg odstępów dla n elementowego zbioru można wyznaczyć następująco:

Przyjmujemy $h_1 = 1$
obliczamy $h_s = 3h_{s-1} + 1$ aż do momentu gdy $h_s >= n$
Ostatecznie $h = [h_s : 9]$

Przykład :
Obliczmy pierwszy odstęp dla zbioru o n = 200 elementach:

$h_1 = 1$
$h_2 = 3h_1 + 1 = 3 + 1 = 4$ - mniejsze od 200, kontynuujemy
$h_3 = 3h_2 + 1 = 12 + 1 = 13$ - kontynuujemy
$h_4 = 3h_3 + 1 = 39 + 1 = 40$ - kontynuujemy
$h_5 = 3h_4 + 1 = 120 + 1 = 121$ - kontynuujemy
$h_6 = 3h_5 + 1 = 360 + 1 = 361$ - stop, ponieważ jest większe od 200
$h = [h_6 : 9] = [361 : 9] = [40\ 1/9] = 40$ (zwróć uwagę, iż jest to zawsze element wcześniejszy o dwie pozycje, czyli $h_4$)

Kolejne odstępy obliczamy dzieląc całkowitoliczbowo bieżący odstęp przez 3. Taki właśnie sposób wyliczania odstępów przyjmiemy w naszym algorytmie.

***Specyfikacja problemu :***

*Dane wejściowe :*

n - liczba elementów w sortowanym zbiorze, $n \in N$
d [ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do n

*Dane wyjściowe :*

d [ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do n

*Zmienne pomocnicze :*

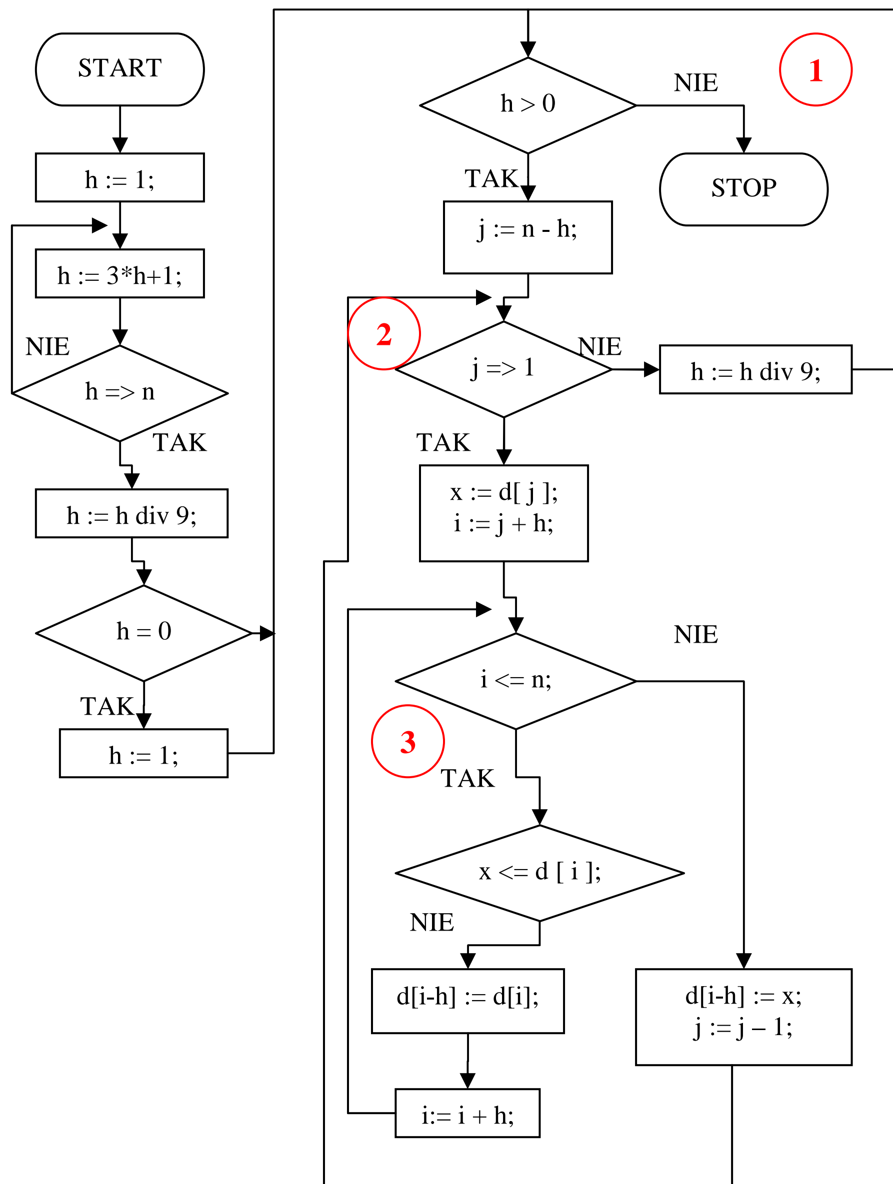
i - indeks elementu listy uporządkowanej, $i \in N$
j - zmienna sterująca pętli, $j \in N$

h - odstęp pomiędzy kolejnymi elementami podzbiorów, $h \in N$
x - zawiera wybrany ze zbioru element

### *Lista kroków*

Krok 1 : h := 1;
Krok 2 : Powtarzaj h := 3*h + 1, aż h >= n
Krok 3 : h := h div 9
Krok 4 : Jeżeli h = 0 to h := 1;
Krok 5 : Dopóki h > 0: wykonuj kroki 6...10
Krok 6 : Dla j = n - h, n - h - 1, ..., 1: wykonuj kroki 7...9
Krok 7 : x := d[j]; i := j + h
Krok 8 : Dopóki (i <= n) i (x > d[i]): wykonuj d[i - h] := d[i]; i := i + h
Krok 9 : d[i - h] := x
Krok 10 : h := h div 3
Krok 11 : Zakończ

### *Schemat blokowy :*




Na początku algorytmu wyznaczamy wartość początkowego odstępu *h*.

Po wyznaczeniu *h* rozpoczynamy pętlę warunkową nr 1. Pętla ta jest wykonywana dotąd, aż odstęp *h* przyjmie wartość 0. Wtedy kończymy algorytm, zbiór będzie posortowany.

Wewnątrz pętli nr 1 umieszczony jest opisany wcześniej algorytm sortowania przez wstawianie, który dokonuje sortowania elementów poszczególnych podzbiorów wyznaczonych przez odstęp *h*. Po zakończeniu sortowania podzbiorów odstęp *h* jest zmniejszany i następuje powrót na początek pętli warunkowej nr 1.

Uwaga : Zwróć uwagę, iż każdy obieg pętli nr 2 sortuje przemiennie jeden element z kolejnych podzbiorów. Najpierw będą to elementy przedostatnie w kolejnych podzbiorach wyznaczonych odstępem h, później wcześniejsze i wcześniejsze. Takie podejście znacząco upraszcza algorytm sortowania.

| **Cechy Algorytmu Sortowania Metodą Shella** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta(n^{1,14})$ |
| klasa złożoności obliczeniowej typowa | $\Theta(n^{1,15})$ |
| klasa złożoności obliczeniowej pesymistyczna | |
| Sortowanie w miejscu | TAK |
| Stabilność | NIE |

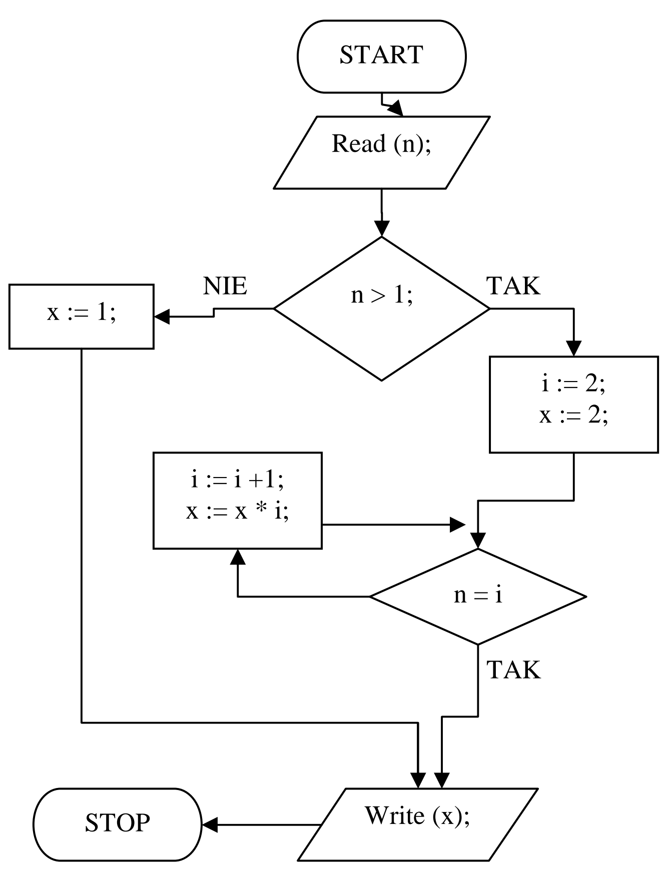
## *8.7. Rekurencja.*

***Zadanie :*** Obliczyć silnię liczby 120 - (120) !

*Rozwiązanie :*

Formalny zapis matematyczny funkcji silnia : $n! = \begin{cases} 1 & , \text{gdy } n \in \{0,1\} \\ 1 \cdot 2 \cdot \ldots \cdot n & , \text{gdy } n \in N_+ - 1 \end{cases}$

Algorytm iteracyjny :




Na początku algorytmu wczytywana jest liczba n, której to mamy obliczyć silnię. Następnie sprawdzany jest warunek czy jest ona większa od 1. Jeśli tak, to licznik i oraz liczbę x (wyliczaną silnię) ustawiamy na 2 i sprawdzamy, czy licznik i jest równy podanej liczbie n. Licznik ten jest zwiększany o 1 oraz wykonywane jest mnożenie x := x*i aż do momentu, gdy i = n.

Zwróćmy uwagę w jaki sposób obliczna jest silnia.
Znając np. wartość 3 ! = 6 łatwo obliczyć wartość 4 ! = 6*4 = 24 , znając wartość 4! = 24 można łatwo obliczyć 5 ! = 24 * 4 = 120 czyli 4! * 5
Ogólnie można to zapisać w postaci : n! = (n-1) ! * n (definicja rekurencyjna)
Proszę zwrócić na odwołanie rekurencyjne – silnia liczby n jest równa wartości silni liczby n-1 pomnozonej przez n. Oczywiście, aby policzyć (n-1) ! musimy znać silnię liczby (n-2) i pomnożyć ją przez (n-1) itp.

Rekurencja - sposób wykonywania obliczeń, w którym wydzielony podprogram wywołuje siebie samego.

Rekurencję można również zastosować do opisu czynności, które nie są obliczeniami. Klasyczny jest już opis jedzenia kaszki podany przez informatyka rosyjskiego Andrieja P. Jerszowa:

Podprogram *Jedz kaszkę*
*sprawdź czy talerz jest pusty*
*nie - weź łyżkę kaszki i jedz kaszkę (dalej).*

Aby funkcja rekurencyjna była poprawna, musi ona mieć ściśle zdefiniowany warunek zakończenia wywołań rekurencyjnych ( w „kaszce” – sprawdź, czy talerz jest pusty ).

***Rekurencyjne oblicznie silni - lista kroków***

Krok 1 : Jeśli n < 2, silnia(n) := 1 i zakończ algorytm
Krok 2 : silnia(n) := n * silnia(n - 1) i zakończ algorytm

Niestety, programu tego nie da się uruchomić w TP dla liczb większych od 8 – wystąpi błąd przepełnienia stosu koprocesora.

Dzięki rekurencji funkcja wyliczająca wartość silni staje się niezwykle prosta. Najpierw sprawdzamy warunek zakończenia rekurencji, tzn. sytuację, gdy wynik dla otrzymanego zestawu danych jest oczywisty. W przypadku silni sytuacja taka wystąpi dla n < 2 - silnia ma wartość 1. Jeśli warunek zakończania rekurencji nie wystąpi, to wartość wyznaczamy za pomocą rekurencyjnego wywołania obliczania silni dla argumentu zmniejszonego o 1. Wynik tego wywołania mnożymy przez n i zwracamy jako wartość silni dla n.

n = 5 silnia(5) = 5 x silnia(4) = 5 x 24 = 120
n = 4 silnia(4) = 4 x silnia(3) = 4 x 6 = 24
n = 3 silnia(3) = 3 x silnia(2) = 3 x 2 = 6
n = 2 silnia(2) = 2 x silnia(1) = 2 x 1 = 2
n = 1 silnia(1) = 1

## 8.8. Sortowanie przez scalanie.

Poczynając od tego miejsca przechodzimy do opisu algorytmów szybkich, tzn. takich, które posiadają klasę czasowej złożoności obliczeniowej równą $\Theta(n \log n)$ lub nawet lepszą.

W informatyce zwykle obowiązuje zasada, iż prosty algorytm posiada dużą złożoność obliczeniową, natomiast algorytm zaawansowany posiada małą złożoność obliczeniową, ponieważ wykorzystuje on pewne własności, dzięki którym szybciej dochodzi do rozwiązania.

Wiele dobrych algorytmów sortujących korzysta z rekurencji, która powstaje wtedy, gdy do rozwiązania problemu algorytm wykorzystuje samego siebie ze zmienionym zestawem danych.

Wynaleziony w 1945 roku przez Johna von Neumanna algorytm sortowania przez scalanie jest algorytmem rekurencyjnym. Wykorzystuje zasadę dziel i zwyciężaj, która polega na podziale zadania głównego na zadania mniejsze dotąd, aż rozwiązanie stanie się oczywiste. Algorytm sortujący dzieli porządkowany zbiór na kolejne połowy dopóki taki podział jest możliwy (tzn. podzbiór zawiera co najmniej dwa elementy). Następnie uzyskane w ten sposób części zbioru rekurencyjnie sortuje tym samym algorytmem. Posortowane części łączy ze sobą za pomocą scalania tak, aby wynikowy zbiór był posortowany.

Podstawową operacją algorytmu jest scalanie dwóch zbiorów uporządkowanych w jeden zbiór również uporządkowany. Operację scalania realizujemy wykorzystując pomocniczy zbiór, w którym będziemy tymczasowo odkładać scalane elementy dwóch zbiorów. Ogólna zasada jest następująca:

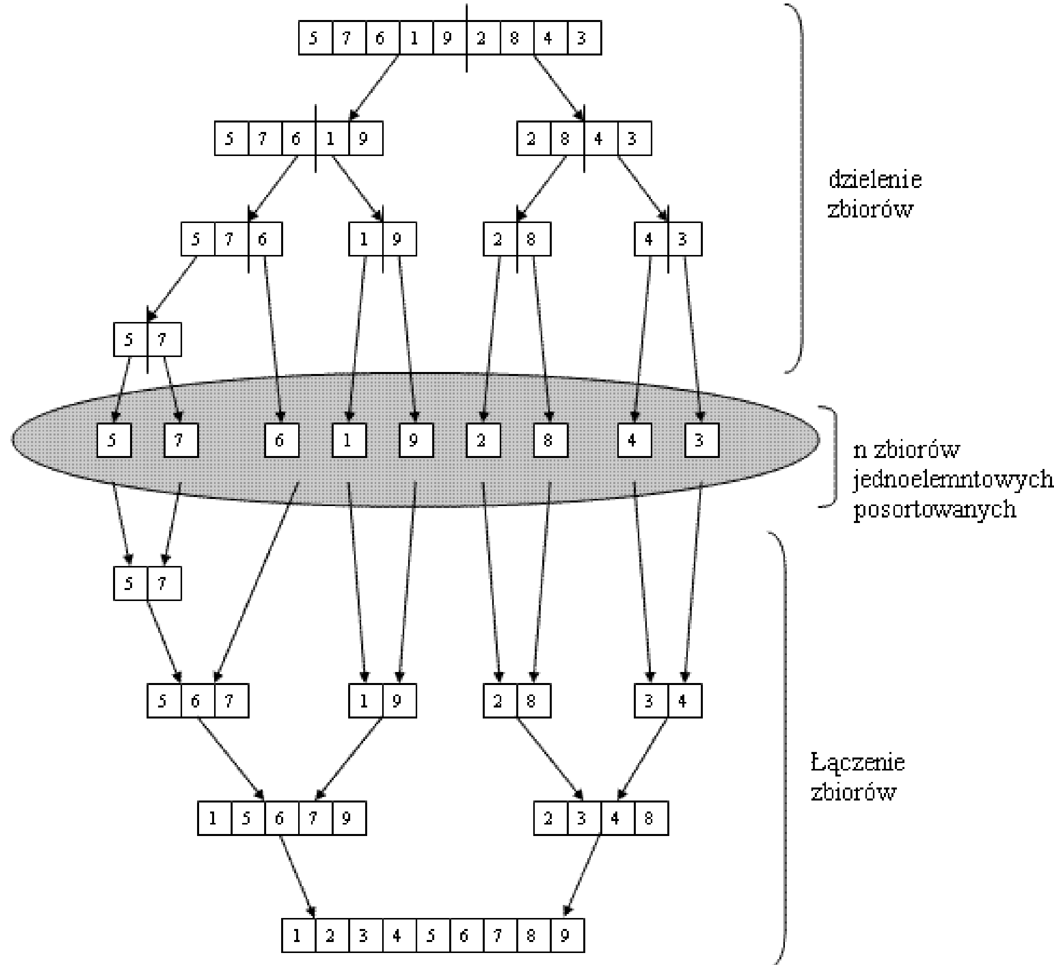
1. Przygotuj pusty zbiór tymczasowy
2. Dopóki żaden ze scalanych zbiorów nie jest pusty, porównuj ze sobą pierwsze elementy każdego z nich i w zbiorze tymczasowym umieszczaj mniejszy z elementów usuwając go jednocześnie ze scalanego zbioru.
3. W zbiorze tymczasowym umieść zawartość tego scalanego zbioru, który zawiera jeszcze elementy.
4. Zawartość zbioru tymczasowego przepisz do zbioru wynikowego i zakończ algorytm.

Przykład : Połączmy za pomocą opisanego algorytmu dwa uporządkowane zbiory: { 1 3 6 7 9 } z { 2 3 4 6 8 }

| Scalane zbiory | Zbiór tymczasowy | Opis wykonywanych działań |
|---|---|---|
| [1] 3 6 7 9<br>2 3 4 6 8 | | Porównujemy ze sobą najmniejsze elementy scalanych zbiorów. Ponieważ zbiory te są już uporządkowane, to najmniejszymi elementami będą zawsze ich pierwsze elementy. |
| 3 6 7 9<br>2 3 4 6 8 | [1] | W zbiorze tymczasowym umieszczamy mniejszy element, w tym przypadku będzie to liczba 1. Jednocześnie element ten zostaje usunięty z pierwszego zbioru |
| 3 6 7 9<br>[2] 3 4 6 8 | 1 | Porównujemy kolejne dwa elementy i mniejszy umieszczamy w zbiorze tymczasowym. |
| [3] 6 7 9<br>3 4 6 8 | 1[2] | Następne porównanie i w zbiorze tymczasowym umieszczamy liczbę 3. Ponieważ są to elementy równe, to nie ma znaczenia, z którego zbioru weźmiemy element 3. |
| 6 7 9<br>[3] 4 6 8 | 1 2[3] | Teraz do zbioru tymczasowego trafi drugie 3. |
| 6 7 9<br>[4] 6 8 | 1 2 3[3] | W zbiorze tymczasowym umieszczamy mniejszy z porównywanych elementów, czyli liczbę 4. |
| [6] 7 9<br>6 8 | 1 2 3 3[4] | Porównywane elementy są równe, zatem w zbiorze tymczasowym umieszczamy dowolny z nich. |
| 7 9<br>[6] 8 | 1 2 3 3 4[6] | Teraz drugą liczbę 6. |

| | | |
|---|---|---|
| [7] 9<br>8 | 1 2 3 3 4 6[6] | W zbiorze tymczasowym umieszczamy liczbę 7 |
| 9<br>[8] | 1 2 3 3 4 6 6[7] | Teraz 8 |
| [9] | 1 2 3 3 4 6 6 7[8] | Drugi zbiór jest pusty. Od tego momentu już nie porównujemy, lecz wprowadzamy do zbioru tymczasowego wszystkie pozostałe elementy pierwszego zbioru, w tym przypadku będzie to liczba 9. |
| | 1 2 3 3 4 6 6 7 8[9] | Zbiór tymczasowy zawiera wszystkie elementy scalanych zbiorów i jest uporządkowany. Przepisujemy jego zawartość do zbioru docelowego. |

Z podanego przykładu możemy wyciągnąć wniosek, iż operacja scalania dwóch uporządkowanych zbiorów jest dosyć prosta.
Tylko jeszcze pozostaje pytanie : jak posortować scalane zbiory ?
Otóż ideą działania algorytmu jest dzielenie zbioru danych na mniejsze zbiory, aż do uzyskania n zbiorów jednoelementowych, które same z siebie są posortowane, następnie zbiory te są łączone w coraz większe zbiory posortowane, aż do uzyskania jednego, posortowanego zbioru n-elementowego. Etap dzielenia nie jest skomplikowany, dzielenie następuje bez sprawdzania jakichkolwiek warunków. Dzięki temu, w przeciwieństwie do algorytmu QuickSort, następuje pełne rozwinięcie wszystkich gałęzi drzewa. Z kolei łączenie zbiorów posortowanych wymaga odpowiedniego wybierania poszczególnych elementów z łączonych zbiorów z uwzględnieniem faktu, że wielkość zbioru nie musi być równa (parzysta i nieparzysta ilość elementów), oraz tego, iż wybieranie elementów z poszczególnych zbiorów nie musi następować naprzemiennie, przez co jeden zbiór może osiągnąć swój koniec wcześniej niż drugi. Robi się to w następujący sposób. Kopiujemy zawartość zbioru głównego do struktury pomocniczej. Następnie, operując wyłącznie na kopii, ustawiamy wskaźniki na początki kolejnych zbiorów i porównujemy wskazywane wartości. Mniejszą wartość wpisujemy do zbioru głównego i przesuwamy odpowiedni wskaźnik o 1 i czynności powtarzamy, aż do momentu, gdy jeden ze wskaźników osiągnie koniec zbioru. Wówczas mamy do rozpatrzenia dwa przypadki, gdy zbiór 1 osiągnął koniec i gdy zbiór 2 osiągnął koniec. W przypadku pierwszym nie będzie problemu, elementy w zbiorze głównym są już posortowane i ułożone na właściwych miejscach. W przypadku drugim trzeba skopiować pozostałe elementy zbioru pierwszego po kolei na koniec. Po zakończeniu wszystkich operacji otrzymujemy posortowany zbiór główny.
Przykład :




Przykład : sortujemy zbiór o postaci: { 6 5 4 1 3 7 9 2 }

| Sortowany zbiór | | | | | | | | Opis wykonywanych operacji |
|---|---|---|---|---|---|---|---|---|
| **d[1]** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** | **d[7]** | **d[8]** | |
| 6 | 5 | 4 | 1 | 3 | 7 | 9 | 2 | Zbiór wyjściowy. |
| 6 | 5 | 4 | 1 | 3 | 7 | 9 | 2 | Pierwszy podział. |
| 6 | 5 | 4 | 1 | 3 | 7 | 9 | 2 | Drugi podział |
| 6 | 5 | 4 | 1 | 3 | 7 | 9 | 2 | Trzeci podział. |
| 5 | 6 | 1 | 4 | 3 | 7 | 2 | 9 | Pierwsze scalanie. |
| 1 | 4 | 5 | 6 | 2 | 3 | 7 | 9 | Drugie scalanie. |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 | Trzecie scalanie. Koniec. |

Ponieważ w opisywanym tutaj algorytmie sortującym scalane podzbiory są przyległymi do siebie częściami innego zbioru, zatem logiczne będzie użycie do ich definicji indeksów wybranych elementów tych podzbiorów:

$i_p$ - **indeks** (nie wartość elementu !) pierwszego elementu w młodszym podzbiorze
$i_s$ - **indeks** pierwszego elementu w starszym podzbiorze
$i_k$ - **indeks** ostatniego elementu w starszym podzbiorze

Przez podzbiór młodszy rozumiemy podzbiór zawierający elementy o indeksach mniejszych niż indeksy elementów w podzbiorze starszym.

Czyli dla powyższego przykładu sytuacja wygląda następująco :

Po pierwszym podziale prezentowanego powyżej zbioru otrzymujemy następujące wartości indeksów:

| **Młodsza połówka** | **Starsza połówka** |
|---|---|
| $i_p = 1$ | $i_s = 5$ |
| $i_k = 8$ | |

Po kolejnym podziale połówek otrzymujemy 4 ćwiartki dwuelementowe. Wartości indeksów będą następujące:

| **Młodsza połówka** | | **Starsza połówka** | |
|---|---|---|---|
| Młodsza ćwiartka | Starsza ćwiartka | Młodsza ćwiartka | Starsza ćwiartka |
| $i_p = 1$ | $i_s = 3$ | $i_p = 5$ | $i_s = 7$ |
| $i_k = 4$ | | $i_k = 8$ | |

***Specyfikacja algorytmu scalania :***

Scalaj (ip, is. ik)

*Dane wejściowe :*
d [ ] - zbiór scalany
ip - indeks pierwszego elementu w młodszym podzbiorze, $ip \in N$
is - indeks pierwszego elementu w starszym podzbiorze, $is \in N$
ik - indeks ostatniego elementu w starszym podzbiorze, $ik \in N$
*Dane wyjściowe :*
d [ ] - zbiór scalony

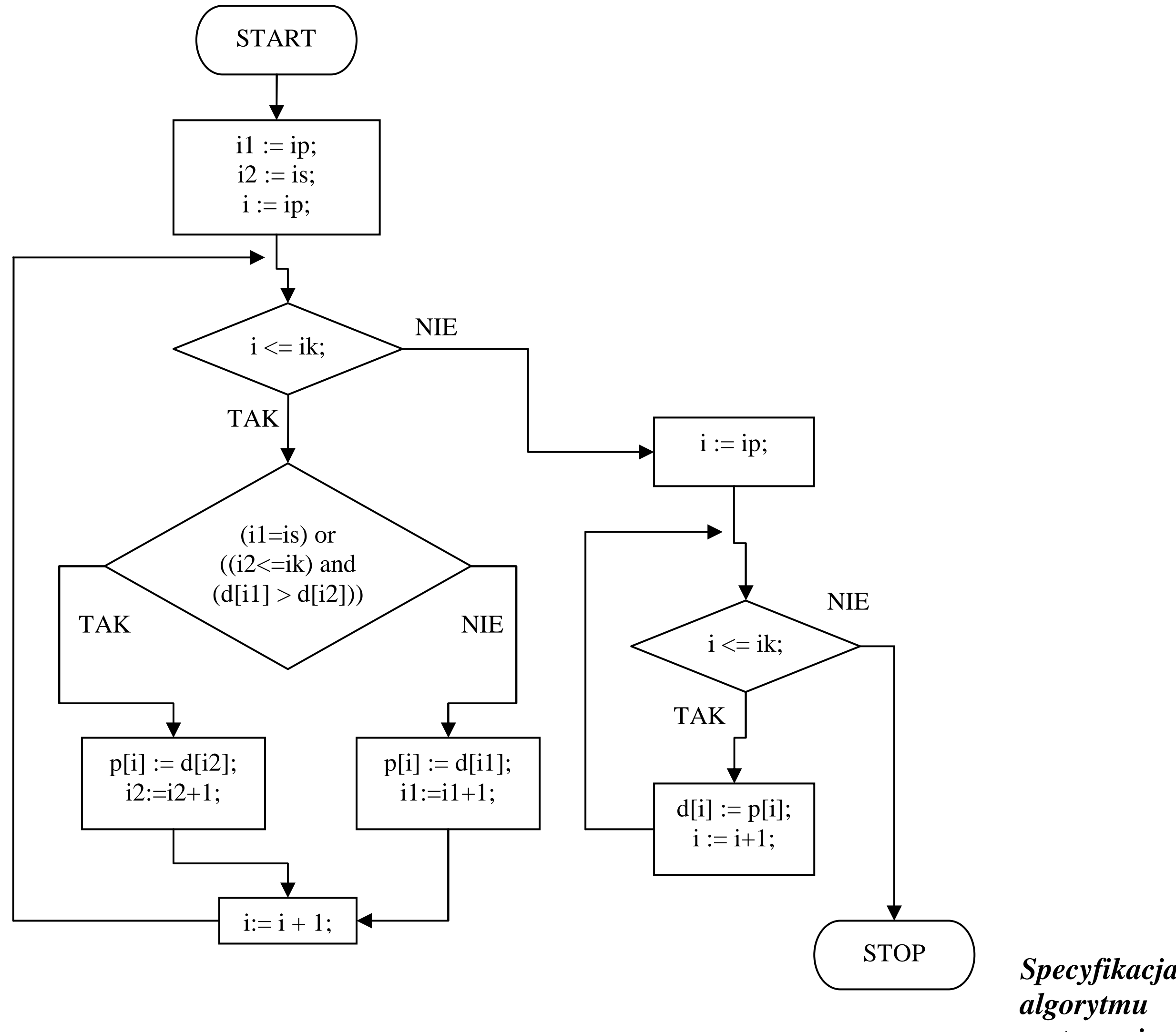
*Dane pomocnicze :*
p [ ] - zbiór pomocniczy zawierający tyle samo elementów ile zbiór d
i1 - indeks elementów w młodszej połówce zbioru d[ ], $i1 \in N$
i2 - indeks elementów w starszej połówce zbioru d[ ], $i2 \in N$
i - indeks elementów w zbiorze p[ ], $i \in N$

***Lista kroków algorytmu scalania***

```
Krok 1 :   i1 := ip;          i2 := is;          i := ip;
Krok 2 :   Dla i:= ip , ip + 1, ........ ik wykonaj :
           Jeśli (i1 =  is) or ((i2 <= ik) and (d[i1] > d[i2])) to
                    p[i] := d[i2] ;          i2:= i2 + 1;
             w przeciwnym przypadku
                    p[i] := d[i1] ;          i1:= i1 + 1;

Krok 3 :   Dla i:= ip , ip + 1, ........ ik wykonaj :  d[i] := p[i];
Krok 4 :   Zakończ algorytm
```

***Schemat blokowy algorytmu scalania***




***Specyfikacja algorytmu sortowania :***

Sortuj (ip,ik)

*Dane wejściowe :*

d [ ] - zbiór scalany
ip - indeks pierwszego elementu w młodszym podzbiorze, $ip \in N$
ik - indeks ostatniego elementu w starszym podzbiorze, $ik \in N$

*Dane wyjściowe :*

d [ ] - zbiór scalony

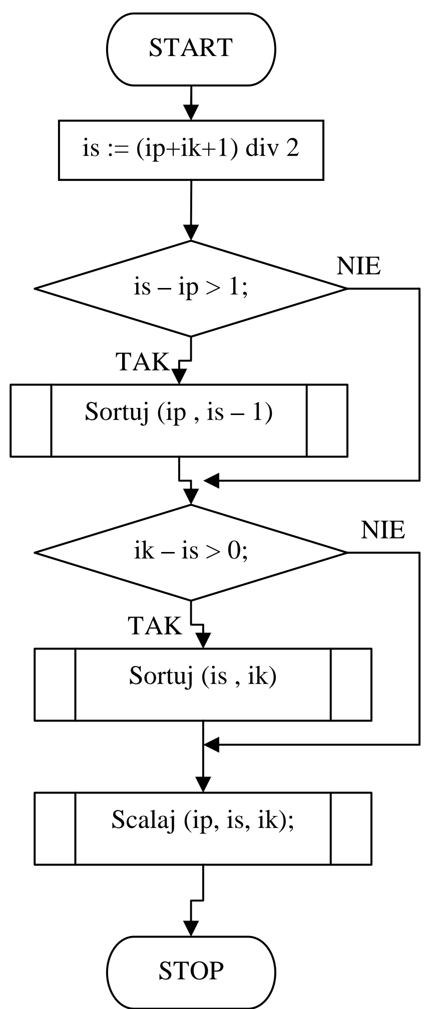
*Dane pomocnicze :*

is - indeks pierwszego elementu w starszym podzbiorze, $is \in N$

***Lista kroków algorytmu sortowania***

Krok 1 : is := (ip + ik +1 ) div 2;
Krok 2 : Jeśli is – ip >1 to wywołanie rekurencyjne Sortuj ( ip, is – 1);
Krok 3 : Jeśli ik – is > 0 to wywołanie rekurencyjne Sortuj ( is, ik );
Krok 4 : wywołanie Scalaj ( ip, is, ik )
Krok 5 : zakończ algorytm

***Schemat blokowy algorytmu sortowania przez scalanie***




Algorytm sortowania przez scalanie jest algorytmem rekurencyjnym. Wywołuje się go z zadanymi wartościami indeksów *ip* oraz *ik*. Przy pierwszym wywołaniu indeksy te powinny objąć cały zbiór d[ ], czyli *ip* = 1, a *ik* = n.
Najpierw algorytm wyznacza indeks *is*, który wykorzystywany jest do podziału zbioru na dwie połówki:
- młodszą o indeksach elementów od *ip* do *is* - 1
- starszą o indeksach elementów od *is* do *ik*

Następnie sprawdzamy, czy dana połówka zbioru zawiera więcej niż jeden element. Jeśli tak, to rekurencyjnie sortujemy ją tym samym algorytmem.

Po posortowaniu obu połówek zbioru scalamy je za pomocą opisanej wcześniej procedury scalania podzbiorów uporządkowanych i kończymy algorytm. Zbiór jest posortowany.

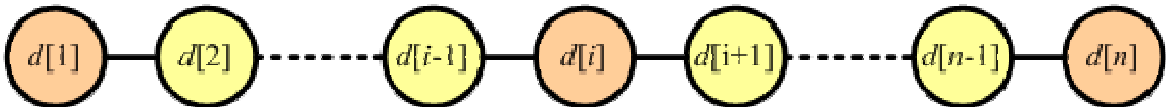
W przykładowym programie procedurę scalania można umieścić bezpośrednio w kodzie algorytmu sortującego, aby zaoszczędzić na wywoływaniu.

| **Cechy Algorytmu Sortowania Przez Scalanie** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | Θ(n log n) |
| klasa złożoności obliczeniowej typowa | |
| klasa złożoności obliczeniowej pesymistyczna | |
| Sortowanie w miejscu | NIE |
| Stabilność | TAK |

## 8.9. Sortowanie stogowe (przez kopcowanie , heap sort) .

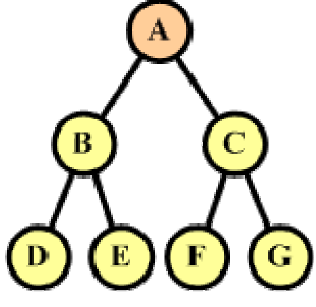
### 8.9.1. Drzewo binarne.

Dotychczas operowaliśmy na prostych strukturach danych, takich jak tablice. W tablicy elementy ułożone są zgodnie z ich numeracją, czyli indeksami. Jeśli za punkt odniesienia weźmiemy element d[i] (i = 2,3,..,n-1; n - liczba elementów w tablicy), to elementem poprzedzającym go będzie element o mniejszym o 1 indeksie, czyli d[i - 1]. Elementem następnym będzie element o indeksie o 1 większym, czyli d[i + 1]. Jest to tzw. hierarchia liniowa - elementy następują jeden za drugim. Graficznie możemy przedstawić to następująco :




Pierwszy element d[1] nie posiada poprzednika (elementu poprzedzającego w ciągu). Ostatni element d[n] nie posiada następnika (elementu następnego w ciągu). Wszystkie pozostałe elementy posiadają poprzedniki i następniki.

Drzewo binarne jest hierarchiczną strukturą danych, którego elementy będziemy nazywali węzłami (ang. node) lub wierzchołkami. W hierarchii liniowej każdy element może posiadać co najwyżej jeden następnik. W drzewie binarnym każdy węzeł może posiadać dwa następniki (stąd pochodzi nazwa drzewa - binarny = dwójkowy, zawierający dwa elementy), które nazwiemy potomkami. dziećmi lub węzłami potomnymi danego węzła (ang. child node).




Węzły są połączone krawędziami symbolizującymi następstwo kolejnych elementów w strukturze drzewa binarnego. Według rysunku po prawej stronie węzeł A posiada dwa węzły potomne: B i C. Węzeł B nosi nazwę lewego potomka (ang. left child node), a węzeł C nosi nazwę prawego potomka (ang. right child node).

Z kolei węzeł B posiada węzły potomne D i E, a węzeł C ma węzły potomne F i G. Jeśli dany węzeł nie posiada dalszych węzłów potomnych, to jest w strukturze drzewa binarnego węzłem terminalnym. Taki węzeł nosi nazwę liścia (ang. leaf node). Na naszym rysunku liściami są węzły terminalne D, E, F i G.

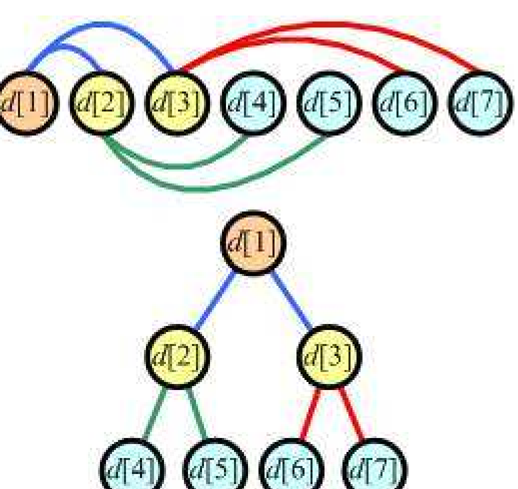
Rodzicem, przodkiem (ang. parent node) lub węzłem nadrzędnym będziemy nazywać węzeł leżący na wyższym poziomie hierarchii drzewa binarnego. Dla węzłów B I C węzłem nadrzędnym jest węzeł A. Podobnie dla węzłów D i E węzłem nadrzędnym będzie węzeł B, a dla F i G będzie to węzeł C.

Węzeł nie posiadający rodzica nazywamy korzeniem drzewa binarnego (ang. root node). W naszym przykładzie korzeniem jest węzeł A. Każde drzewo binarne, które zawiera węzły posiada dokładnie jeden korzeń.

Jeśli chcemy przetwarzać za pomocą komputera struktury drzew binarnych, to musimy zastanowić się nad sposobem reprezentacji takich struktur w pamięci. Najprostszym rozwiązaniem jest zastosowanie zwykłej tablicy n elementowej. Każdy element tej tablicy będzie reprezentował jeden węzeł drzewa binarnego. Pozostaje nam jedynie określenie związku pomiędzy indeksami elementów w tablicy a położeniem tych elementów w strukturze drzewa binarnego.

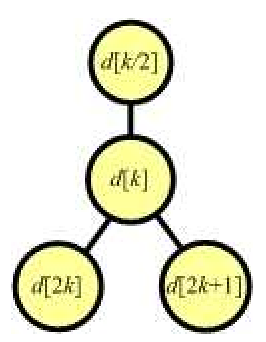
Zastosujmy następujące odwzorowanie:

- Element d[1] będzie zawsze korzeniem drzewa.
- i-ty poziom drzewa binarnego wymaga $2^{i-1}$ węzłów. Będziemy je kolejno pobierać z tablicy.




Otrzymamy w ten sposób następujące odwzorowanie elementów tablicy w drzewo binarne:

Dla węzła k-tego wyprowadzamy następujące wzory:

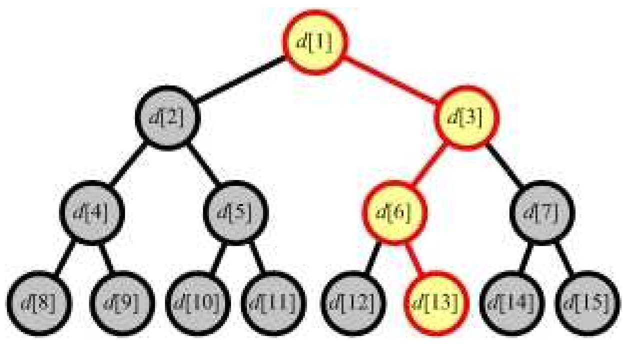



węzły potomne mają indeksy równe:
*2k* - lewy potomek
*2k+1* - prawy potomek
węzeł nadrzędny ma indeks równy *[k / 2]* (dzielenie całkowitoliczbowe)

Przykład : Skonstruować drzewo binarne z elementów zbioru {7 5 9 2 4 6 1}

| Operacja | Opis |
|---|---|
| 7<br>**7** 5 9 2 4 6 1 | Konstrukcję drzewa binarnego rozpoczynamy od korzenia, który jest pierwszym elementem zbioru, czyli liczbą 7. |
| 7, 5, 9<br>7 **5 9** 2 4 6 1 | Do korzenia dołączamy dwa węzły potomne, które leżą obok w zbiorze. Są to dwa kolejne elementy, 5 i 9. |
| 7, 5, 9, 2, 4<br>7 5 9 **2 4** 6 1 | Do lewego węzła potomnego (5) dołączamy jego węzły potomne. Są to kolejne liczby w zbiorze, czyli 2 i 4. |
| 7, 5, 9, 2, 4, 6, 1<br>7 5 9 2 4 **6 1** | Pozostaje nam dołączyć do prawego węzła ostatnie dwa elementy zbioru, czyli liczby 6 i 1. Drzewo jest kompletne. |

**Ścieżką** nazywamy ciąg węzłów drzewa binarnego spełniających warunek, iż każdy węzeł poprzedni jest rodzicem węzła następnego. Jeśli ścieżka składa się z k węzłów, to długością ścieżki jest liczba k - 1.




Na rysunku obok została zaznaczona ścieżka biegnąca poprzez węzły {d[1], d[3], d[6], d[13]}. Ścieżka ta zawiera cztery węzły, ma zatem długość równą 3.
Wysokością drzewa binarnego nazwiemy długość najdłuższej ścieżki od korzenia do liścia. W powyższym przykładzie najdłuższa taka ścieżka ma długość 3, zatem zaprezentowane drzewo binarne ma wysokość równą 3.
Dla n węzłów zrównoważone drzewo binarne ma wysokość równą: $h = [\log_2 n]$

węzeł (ang. node) - element drzewa binarnego
rodzic, węzeł nadrzędny, przodek (ang, parent node) - węzeł leżący o 1 poziom wyżej w hierarchii
dziecko, potomek, węzeł potomny (ang. child node) - węzeł leżący o 1 poziom niżej w hierarchii, dla którego bieżący węzeł jest rodzicem.
korzeń drzewa (ang. root node) - pierwszy węzeł na najwyższym poziomie hierarchii, który nie posiada rodzica
liść, węzeł terminalny (ang. leaf node) - węzeł nie posiadający węzłów potomnych
ścieżka (ang. path) - droga na drzewie binarnym wiodąca poprzez poszczególne wierzchołki

### 8.9.2. Tworzenie kopca

Kopiec jest drzewem binarnym, w którym wszystkie węzły spełniają następujący warunek (zwany warunkiem kopca) : węzeł nadrzędny jest większy lub równy węzłom potomnym (w porządku malejącym relacja jest odwrotna - mniejszy lub równy).

Konstrukcja kopca jest nieco bardziej skomplikowana od konstrukcji drzewa binarnego, ponieważ musimy dodatkowo troszczyć się o zachowanie warunku kopca. Zatem po każdym dołączeniu do kopca nowego węzła, sprawdzamy odpowiednie warunki i ewentualnie dokonujemy wymian węzłów na ścieżce wiodącej od dodanego węzła do korzenia.

***Przykład :*** Skonstruować kopiec z elementów zbioru {7 5 9 6 7 8 10}

| Operacja | Opis |
|---|---|
| 7<br>7 5 9 6 7 8 10 | Budowę kopca rozpoczynamy od pierwszego elementu zbioru, który staje się korzeniem. |
| 7, 5<br>7 5 9 6 7 8 10 | Do korzenia dołączamy następny element. Warunek kopca jest zachowany. |

| Operacja | Opis |
|---|---|
| 7 5 **9** 6 7 8 10 | Dodajemy kolejny element ze zbioru. |
| | Po dodaniu elementu 9 warunek kopca przestaje być spełniony. Musimy go przywrócić. W tym celu za nowy węzeł nadrzędny wybieramy nowo dodany węzeł. Poprzedni węzeł nadrzędny wędruje w miejsce węzła dodanego - zamieniamy węzły 7 i 9 miejscami. |
| | Po wymianie węzłów 7 i 9 warunek kopca jest spełniony. |
| 7 5 9 **6** 7 8 10 | Dołączamy kolejny element 6. Znów warunek kopca przestaje być spełniony - zamieniamy miejscami węzły 5 i 6. |
| | Po wymianie węzłów 5 i 6 warunek kopca jest spełniony. |
| 7 5 9 6 **7** 8 10 | Dołączamy kolejny element 7. Warunek kopca przestaje obowiązywać. Zamieniamy miejscami węzły 6 i 7. |
| | Po wymianie węzłów 6 i 7 warunek kopca obowiązuje. |
| 7 5 9 6 7 **8** 10 | Dołączamy kolejny węzeł. Powoduje on naruszenie warunku kopca, zatem wymieniamy ze sobą węzły 7 i 8. |
| | Po wymianie węzłów 7 i 8 warunek kopca znów obowiązuje. |

| Operacja | Opis |
| --- | --- |
| 7 5 9 6 7 8 10 | Dołączenie ostatniego elementu znów narusza warunek kopca. Zamieniamy miejscami węzeł 8 z węzłem 10. |
| | Po wymianie węzłów 8 i 10 warunek kopca został przywrócony na tym poziomie. Jednakże węzeł 10 stał się dzieckiem węzła 9. Na wyższym poziomie drzewa warunek kopca jest naruszony. Aby go przywrócić znów wymieniamy miejscami węzły, tym razem węzeł 9 z węzłem 10. |
| | Po wymianie tych węzłów warunek kopca obowiązuje w całym drzewie - zadanie jest wykonane. |

Charakterystyczną cechą kopca jest to, iż korzeń zawsze jest największym (w porządku malejącym najmniejszym) elementem z całego drzewa binarnego

***Specyfikacja algorytmu konstrukcji kopca.***

*Dane wejściowe :*

d [ ] - Zbiór zawierający elementy do wstawienia do kopca. Numeracja elementów rozpoczyna się od 1

n - Ilość elementów w zbiorze, $n \in N$

*Dane wyjściowe :*

d [ ] - zbiór zawierający kopiec

*Zmienne pomocnicze :*

i - zmienna licznikowa pętli umieszczającej kolejne elementy zbioru w kopcu, $i \in N$, $i \in \{2,3,...,n\}$

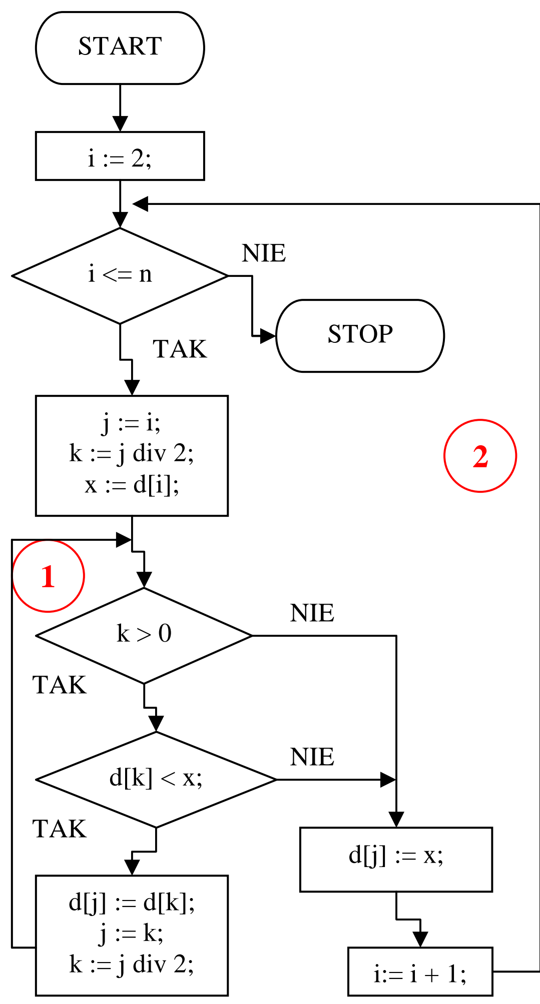
j,k - indeksy elementów leżących na ścieżce od wstawianego elementu do korzenia, $j,k \in C$

x - zmienna pomocnicza przechowująca tymczasowo element wstawiany do kopca

***Lista kroków :***

Krok 1 : Dla i = 2, ..., n: wykonuj kroki 2...5
Krok 2 : j := i; k := j div 2;
Krok 3 : x := d [ i ];
Krok 4 : Dopóki ( k > 0 ) and ( d [k] < x ) wykonuj :
    d [ j ] := d [ k ];
    j := k;
    k := j div 2;
Krok 5 : d [ j ] := x;
Krok 6 : koniec

*Schemat blokowy*




Algorytm tworzy kopiec w tym samym zbiorze wejściowym d[ ]. Nie wymaga zatem dodatkowych struktur danych i ma złożoność pamięciową klasy Θ(n).
Pętla nr 1 wyznacza kolejne elementy wstawiane do kopca. Pierwszy element pomijamy, ponieważ zostałby i tak na swoim miejscu. Dlatego pętla rozpoczyna wstawianie od elementu nr 2.
Wewnątrz pętli nr 1 inicjujemy kilka zmiennych:
j - pozycja wstawianego elementu (liścia)
k - pozycja elementu nadrzędnego (przodka)
x - zapamiętuje wstawiany element

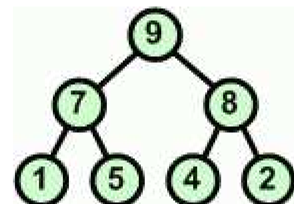
Następnie rozpoczynamy pętlę warunkową nr 2, której zadaniem jest znalezienie w kopcu miejsca do wstawienia zapamiętanego elementu w zmiennej x. Pętla ta wykonuje się do momentu osiągnięcia korzenia kopca (k = 0) lub znalezienia przodka większego od zapamiętanego elementu. Wewnątrz pętli przesuwamy przodka na miejsce potomka, aby zachować warunek kopca, a następnie przesuwamy pozycję j na pozycję zajmowaną wcześniej przez przodka. Pozycja k staje się pozycją nowego przodka i pętla się kontynuuje. Po jej zakończeniu w zmiennej j znajduje się numer pozycji w zbiorze d[ ], na której należy umieścić element w x.
Po zakończeniu pętli nr 1 w zbiorze zostaje utworzona struktura kopca.

### 8.9.3. Rozbiór kopca.

Zasady rozbioru kopca :

1. Zamień miejscami korzeń z ostatnim liściem, który wyłącz ze struktury kopca. Elementem pobieranym z kopca jest zawsze jego korzeń, czyli element największy.
2. Jeśli jest to konieczne, przywróć warunek kopca idąc od korzenia w dół.
3. Kontynuuj od kroku 1, aż kopiec będzie pusty.

Przykład : Rozebrać kopiec



| Operacja | Opis |
| --- | --- |
| 9 | Rozbiór kopca rozpoczynamy od korzenia, który usuwamy ze struktury kopca. W miejsce korzenia wstawiamy ostatni liść. |
| | Poprzednia operacja zaburzyła strukturę kopca. Idziemy zatem od korzenia w dół struktury przywracając warunek kopca - przodek większy lub równy od swoich potomków. Praktycznie polega to na zamianie przodka z największym potomkiem. Operację kontynuujemy dotąd, aż natrafimy na węzły spełniające warunek kopca. |
| 8 9 | Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem |
| | W nowym kopcu przywracamy warunek kopca. |
| 7 8 9 | Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem |
| | W nowym kopcu przywracamy warunek kopca. W tym przypadku już po pierwszej wymianie węzłów warunek koca jest zachowany w całej strukturze. |

| Operacja | Opis |
| --- | --- |
| 5 7 8 9 | Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem |
| | Przywracamy warunek kopca w strukturze. |
| 4 5 7 8 9 | Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem |
| | Przywracamy warunek kopca w strukturze. |
| 2 4 5 7 8 9 | Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem. |
| 1 2 4 5 7 8 9 | Po wykonaniu poprzedniej operacji usunięcia w kopcu pozostał tylko jeden element - usuwamy go. Zwróć uwagę, iż usunięte z kopca elementy tworzą ciąg uporządkowany. |

Operację rozbioru kopca można przeprowadzić w miejscu. Jeśli mamy zbiór d[ ] ze strukturą kopca, to idąc od końca zbioru (ostatnie liście drzewa) w kierunku początku zamieniamy elementy z pierwszym elementem zbioru (korzeń drzewa), a następnie poruszając się od korzenia w dół przywracamy warunek kopca w kolejno napotykanych węzłach.

***Specyfikacja algorytmu rozbioru kopca.***

*Dane wejściowe :*

d [ ] - Zbiór zawierający poprawną strukturę kopca. Numeracja elementów rozpoczyna się od 1

n - Ilość elementów w zbiorze, $n \in N$

*Dane wyjściowe :*

d [ ] - Zbiór zawierający elementy pobrane z kopca ułożone w porządku rosnącym

*Zmienne pomocnicze :*

i - indeksy elementów leżących na ścieżce w dół od korzenia, $i \in N$, $i \in \{n, n-1, ....2\}$

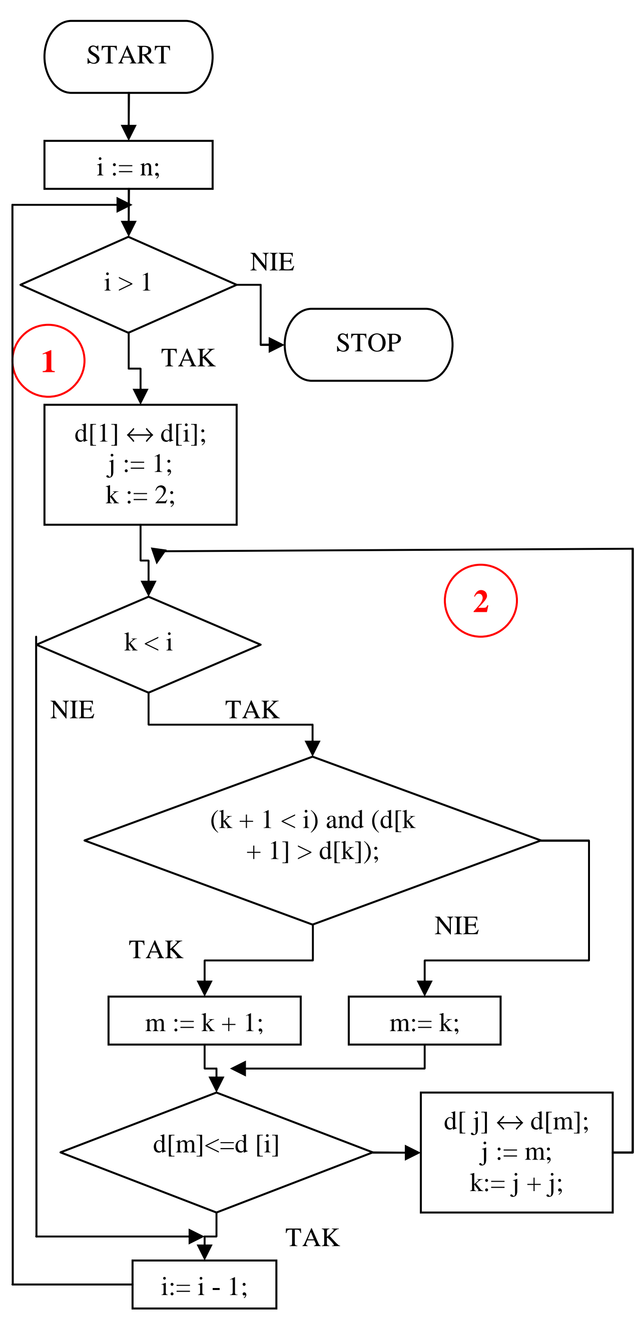
j,k - indeksy elementów leżących na ścieżce w dół od korzenia, $j,k \in N$

m - indeks większego z dwóch elementów potomnych, $m \in N$,

***Lista kroków :***

Krok 1 : Dla i = n, n-1 , ..., 2: wykonuj kroki 2...8
Krok 2 : d [1] ↔ d[ i ];
Krok 3 : j := 1; k := 2;
Krok 4 : Dopóki (k < i) wykonuj kroki 6...8
Krok 5 : Jeżeli (k + 1 < i) and (d[k + 1] > d[k]), to m := k + 1. Inaczej m := k
Krok 6 : Jeżeli d[m]<= d[j], to wyjdź z bieżącej pętli i kontynuuj następny obieg pętli z kroku 1
Krok 7 : d[ j ] ↔ d[m]
Krok 8 : j := m; k := j + j i kontynuuj pętlę z kroku 4
Krok 9 : Zakończ algorytm

***Schemat blokowy***




Rozbiór kopca wykonywany jest w dwóch zagnieżdżonych pętlach. Pętla nr 1 zamienia miejscami kolejne liście ze spodu drzewa z korzeniem. Zadaniem pętli nr 2 jest przywrócenie w strukturze warunku kopca.
Pętla nr 1 przebiega wstecz indeksy elementów zbioru d[ ] poczynając od indeksu n a kończąc na indeksie 2. Element i-ty jest wymieniany z korzeniem kopca. Po tej wymianie rozpoczynamy w pętli nr 2 przeglądanie drzewa od korzenia w dół. Zmienna j przechowuje indeks przodka, zmienna k przechowuje indeks lewego potomka. Pętla nr 2 kończy się, gdy węzeł j-ty nie posiada potomków. Wewnątrz pętli wyznaczamy w zmiennej m indeks większego z dwóch węzłów potomnych. Następnie sprawdzamy, czy w węźle nadrzędnym j-tym jest zachowany warunek kopca. Jeśli tak, to następuje wyjście z pętli nr 2. W przeciwnym razie zamieniamy miejscami węzeł nadrzędny j-ty z jego największym potomkiem m-tym i za nowy węzeł nadrzędny przyjmujemy węzeł m-ty. W zmiennej k wyznaczamy indeks jego lewego potomka i kontynuujemy pętlę nr 2.
Po wyjściu z pętli nr 2 zmniejszamy zmienną i o 1 - przenosimy się do kolejnego, ostatniego liścia drzewa i kontynuujemy pętlę nr 1.
W celu wyznaczenia klasy złożoności obliczeniowej algorytmu rozbioru kopca zauważamy, iż pętla zewnętrzna nr 1 wykona się n - 1 razy, a w każdym

obiegu tej pętli pętla wewnętrzna wykona się maksymalnie $\log_2$ n razy. Daje to zatem górne oszacowanie Θ(n log n) w przypadku pesymistycznym oraz Θ (n) w przypadku optymistycznym, który wystąpi tylko wtedy, gdy zbiór d[ ] będzie zbudowany z ciągu tych samych elementów.

Zwróćmy uwagę, że jeśli w zbiorze utworzymy kopiec, który następnie rozbierzemy to w wyniku tych operacji otrzymamy zbiór posortowany.

***Lista kroków :***

Krok 1 : Tworz_Kopiec
Krok 2 : Rozbierz_Kopiec
Krok 3 : Zakończ algorytm

| **Cechy Algorytmu Sortowania Przez Kopcowanie** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | Θ (n log n) |
| klasa złożoności obliczeniowej typowa | |
| klasa złożoności obliczeniowej pesymistyczna | |
| Sortowanie w miejscu | TAK |
| Stabilność | NIE |

Ponieważ sortowanie przez kopcowanie składa się z dwóch następujących bezpośrednio po sobie operacji o klasie czasowej złożoności obliczeniowej Θ(n log n), to dla całego algorytmu klasa złożoności również będzie wynosić Θ (n log n).

## 8.10. Sortowanie szybkie.

Algorytm sortowania szybkiego opiera się na strategii "dziel i zwyciężaj" (ang. divide and conquer), którą możemy krótko scharakteryzować w trzech punktach:
1. DZIEL - problem główny zostaje podzielony na podproblemy
2. ZWYCIĘŻAJ - znajdujemy rozwiązanie podproblemów
3. POŁĄCZ - rozwiązania podproblemów zostają połączone w rozwiązanie problemu głównego

Idea sortowania szybkiego jest następująca:

DZIEL - najpierw sortowany zbiór dzielimy na dwie części w taki sposób, aby wszystkie elementy leżące w pierwszej części (zwanej **lewą partycją**) były mniejsze lub równe od wszystkich elementów drugiej części zbioru (zwanej **prawą partycją**).
ZWYCIĘŻAJ - każdą z partycji sortujemy rekurencyjnie tym samym algorytmem
POŁĄCZ - połączenie tych dwóch partycji w jeden zbiór daje w wyniku zbiór posortowany

Sortowanie szybkie zostało wynalezione przez angielskiego informatyka, profesora Tony'ego Hoare'a w latach 60-tych ubiegłego wieku. W przypadku typowym algorytm ten jest najszybszym algorytmem sortującym z klasy złożoności obliczeniowej Θ(n log n) - stąd pochodzi jego popularność w zastosowaniach. Musimy jednak pamiętać, iż w pewnych sytuacjach (zależnych od sposobu wyboru piwotu oraz niekorzystnego ułożenia danych wejściowych) klasa złożoności obliczeniowej tego algorytmu może się degradować do Θ($n^2$), co więcej, poziom wywołań

rekurencyjnych może spowodować przepełnienie stosu i zablokowanie komputera. Z tych powodów algorytmu sortowania szybkiego nie można stosować bezmyślnie w każdej sytuacji tylko dlatego, iż jest uważany za jeden z najszybszych algorytmów sortujących - zawsze należy przeprowadzić analizę możliwych danych wejściowych właśnie pod kątem przypadku niekorzystnego - czasem lepszym rozwiązaniem może być zastosowanie wcześniej opisanego algorytmu sortowania przez kopcowanie, który nigdy nie degraduje się do klasy $\Theta(n^2)$.

***Tworzenie partycji***

Do utworzenia partycji musimy ze zbioru wybrać jeden z elementów, który nazwiemy **piwotem**. W lewej partycji znajdą się wszystkie elementy niewiększe od piwotu, a w prawej partycji umieścimy wszystkie elementy niemniejsze od piwotu. Położenie elementów równych nie wpływa na proces sortowania, zatem mogą one występować w obu partycjach. Również porządek elementów w każdej z partycji nie jest ustalony.

Jako piwot można wybierać element pierwszy, środkowy, ostatni, medianę lub losowy. Dla naszych potrzeb wybierzemy element środkowy:

piwot := d[(lewy + prawy) **div** 2]

piwot - element podziałowy
d[ ] - dzielony zbiór
lewy - indeks pierwszego elementu
prawy - indeks ostatniego elementu

Dzielenie na partycje polega na umieszczeniu dwóch wskaźników na początku zbioru - i oraz j. Wskaźnik i przebiega przez zbiór poszukując wartości mniejszych od piwotu. Po znalezieniu takiej wartości jest ona wymieniana z elementem na pozycji j. Po tej operacji wskaźnik j jest przesuwany na następną pozycję. Wskaźnik j zapamiętuje pozycję, na którą trafi następny element oraz na końcu wskazuje miejsce, gdzie znajdzie się piwot. W trakcie podziału piwot jest bezpiecznie przechowywany na ostatniej pozycji w zbiorze.

Przykład : Podzielmy na partycje zbiór { 7 ,2 ,4 ,7 ,3 ,1 ,4 ,6 ,5 ,8 ,3 ,9 ,2 ,6 ,7 ,6 ,3 }

| Lp. | Operacja | Opis |
|---|---|---|
| *1.* | 7 2 4 7 3 1 4 6<br>5 8 3 9 2 6 7 6<br>3 | Wyznaczamy na piwot element środkowy. |
| *2.* | 7 2 4 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5 | Piwot wymieniamy z ostatnim elementem zbioru |
| *3.* | 7 2 4 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Na początku zbioru ustawiamy dwa wskaźniki. Wskaźnik „i” będzie przeglądał zbiór do przedostatniej pozycji. Wskaźnik „j” zapamiętuje miejsce wstawiania elementów mniejszych od piwotu |
| *4.* | 7 2 4 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Wskaźnikiem „i” szukamy elementu mniejszego od piwotu |
| *5.* | 2 7 4 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Znaleziony element wymieniamy z elementem na pozycji j-tej. Po wymianie wskaźnik j przesuwamy o 1 pozycję. |

| Lp. | Operacja | Opis |
|---|---|---|
| *6.* | 2 7 4 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Szukamy |
| *7.* | 2 4 7 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Wymieniamy „i” przesuwamy „j”. |
| *8.* | 2 4 7 7 3 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Szukamy |
| *9.* | 2 4 3 7 7 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Wymieniamy „i” przesuwamy „j”. |
| *10.* | 2 4 3 7 7 1 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Szukamy |
| *11.* | 2 4 3 1 7 7 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Wymieniamy „i” przesuwamy „j”. |
| *12.* | 2 4 3 1 7 7 4 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Szukamy |
| *13.* | 2 4 3 1 4 7 7 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Wymieniamy „i” przesuwamy „j”. |
| *14.* | 2 4 3 1 4 7 7 6<br>3 8 3 9 2 6 7 6<br>5<br>i<br>j | Szukamy |
| *15.* | 2 4 3 1 4 3 7 6<br>7 8 3 9 2 6 7 6<br>5<br>i<br>j | Wymieniamy „i” przesuwamy „j”. |

| Lp. | Operacja | Opis |
|---|---|---|
| *16.* | `2 4 3 1 4 3 7 6 7 8 3 9 2 6 7 6`<br>`5`<br>`i`<br>`j` | Szukamy |
| *17.* | `2 4 3 1 4 3 3 6 7 8 7 9 2 6 7 6`<br>`5`<br>`i`<br>`j` | Wymieniamy „i” przesuwamy „j”. |
| *18.* | `2 4 3 1 4 3 3 6 7 8 7 9 2 6 7 6`<br>`5`<br>`i`<br>`j` | Szukamy |
| *19.* | `2 4 3 1 4 3 3 2 7 8 7 9 6 6 7 6`<br>`5`<br>`i`<br>`j` | Wymieniamy „i” przesuwamy „j”. |
| *20.* | `2 4 3 1 4 3 3 2 5 8 7 9 6 6 7 6 7`<br>`^ i`<br>`Lewa partycja`<br>`j Prawa partycja` | Brak dalszych elementów do wymiany. Piwot wymieniamy z elementem na pozycji j-tej. Podział na partycje zakończony. |

Po zakończeniu podziału na partycje wskaźnik j wyznacza pozycję piwotu. Lewa partycja zawiera elementy mniejsze od piwotu i rozciąga się od początku zbioru do pozycji j - 1. Prawa partycja zawiera elementy większe lub równe piwotowi i rozciąga się od pozycji j + 1 do końca zbioru. Operacja podziału na partycje ma liniową klasę złożoności obliczeniowej - $\Theta(n)$.

***Specyfikacja algorytmu.***

*Dane wejściowe :*
d [ ] - Zbiór zawierający elementy do posortowania. Zakres indeksów elementów jest dowolny
lewy - indeks pierwszego elementu w zbiorze, lewy $\in$ C
prawy - indeks ostatniego elementu w zbiorze, prawy $\in$ C

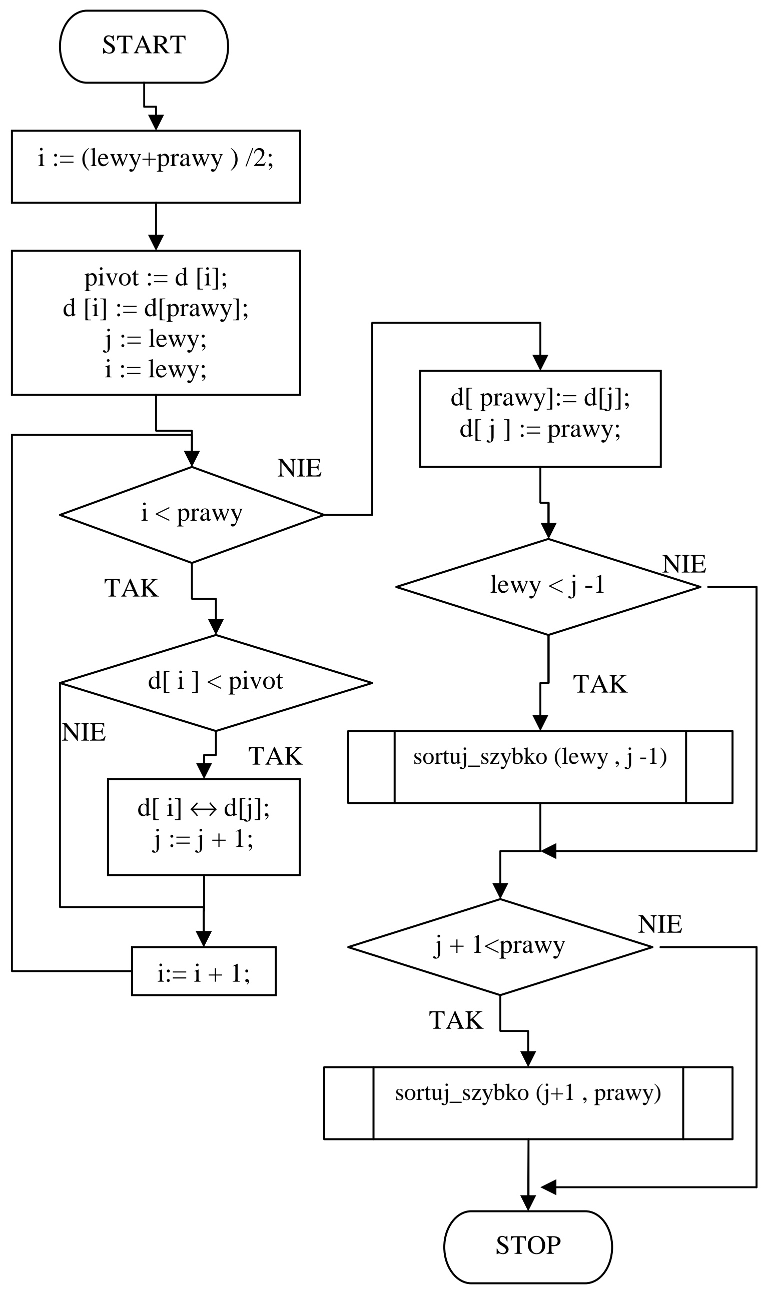
*Dane wyjściowe :*
d [ ] - Zbiór zawierający elementy posortowane

*Zmienne pomocnicze :*
pivot - element podziałowy

i, j - indeksy elementów, i, j ∈ C

***Lista kroków :***

Krok 1 : i := (lewy + prawy) / 2;
Krok 2 : piwot := d[i]; d[i] := d[prawy]; j := lewy
Krok 3 : Dla i = lewy, lewy + 1, ..., prawy - 1: wykonuj kroki 4...5
Krok 4 : Jeśli d[i] >= piwot, to wykonaj kolejny obieg pętli z kroku 3
Krok 5 : d[i] ↔ d[j]; j := j + 1
Krok 6 : d[prawy] := d[j]; d[j] := piwot;
Krok 7 : Jeśli lewy < j - 1, to wykonaj rekurencyjnie *Sortuj_szybko*(*lewy, j* - 1)
Krok 8 : Jeśli j + 1 < prawy, to wykonaj rekurencyjnie *Sortuj_szybko*(*j* + 1, *prawy*)
Krok 9 : Zakończ algorytm




Algorytm sortowania szybkiego wywołujemy podając za lewy indeks pierwszego elementu zbioru, a za prawy indeks elementu ostatniego (czyli Sortuj_szybko(1,n)). Zakres indeksów jest dowolny - dzięki temu ten sam algorytm może również sortować fragment zbioru, co wykorzystujemy przy sortowaniu wyliczonych partycji.

***Schemat blokowy***

Na element podziałowy wybieramy element leżący w środku dzielonej partycji. Wyliczamy jego pozycję i zapamiętujemy ją tymczasowo w zmiennej i. Robimy to po to, aby dwukrotnie nie wykonywać tych samych rachunków.
Element d[i] zapamiętujemy w zmiennej piwot, a do d[i] zapisujemy ostatni element partycji. Dzięki tej operacji piwot został usunięty ze zbioru.
Ustawiamy zmienną j na początek partycji. Zmienna ta zapamiętuje pozycję podziału partycji.
W pętli sterowanej zmienną i przeglądamy kolejne elementy od pierwszego do przedostatniego (ostatni został umieszczony na pozycji piwotu, a piwot zapamiętany). Jeśli i-ty element

jest mniejszy od piwotu, to trafia on na początek partycji - wymieniamy ze sobą elementy na pozycjach i-tej i j-tej. Po tej operacji przesuwamy punkt podziałowy partycji j.
Po zakończeniu pętli element z pozycji j-tej przenosimy na koniec partycji, aby zwolnić miejsce dla piwotu, po czym wstawiamy tam piwot. Zmienna j wskazuje zatem wynikową pozycję piwotu. Pierwotna partycja została podzielona na dwie partycje:
* *partycja lewa* od pozycji lewy do j - 1 zawiera elementy mniejsze od piwotu
* *partycja prawa* od pozycji j + 1 do pozycji prawy zawiera elementy większe lub równe piwotowi.
Sprawdzamy, czy partycje te obejmują więcej niż jeden element. Jeśli tak, to wywołujemy rekurencyjnie algorytm sortowania szybkiego przekazując mu granice wyznaczonych partycji. Po powrocie z wywołań rekurencyjnych partycja wyjściowa jest posortowana rosnąco. Kończymy algorytm.

| **Cechy Algorytmu Sortowania Szybkiego** | |
|---|---|
| klasa złożoności obliczeniowej optymistyczna | $\Theta(n \log n)$ |
| klasa złożoności obliczeniowej typowa | |
| klasa złożoności obliczeniowej pesymistyczna | $\Theta(n^2)$ |
| Sortowanie w miejscu | TAK |
| Stabilność | NIE |

## *8.11. Sortowanie dystrybucyjne.*

Wszystkie opisane dotychchzas algorytmy sortujące opierają się na sprawdzaniu uporządkowania zbioru, które polega na porównywaniu elementów. Udowodniono, iż barierą efektywności takich algorytmów w przypadku ogólnym (sortowanie zbioru o losowym rozkładzie elementów) jest klasa złożoności obliczeniowej $\Theta(n \log n)$. Zachodzi zatem naturalne pytanie: czy istnieją inne sposoby sortowania o niższej klasie złożoności obliczeniowej?

### *8.11.1. Sortowanie rozrzutowe.*

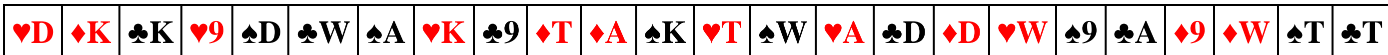
Sortowanie rozrzutowe wyjaśnimy na przykładzie porządkowania talii kart.
Ustalmy kolejność kolorów kart wg ich starszeństwa (pierwszy pik, ostatni trefl):

♠ - pik ♥ - kier ♦ - karo ♣ - trefl

Teraz ustalmy kolejność figur (dla uproszczenia przyjmujemy talię z 24 kart, chociaż algorytm jest również poprawny dla pełnej talii 52 kart):

**A** - as
**K** - król
**D** - dama
**W** - walet
**T** - dziesiątka
**9** - dziewiątka

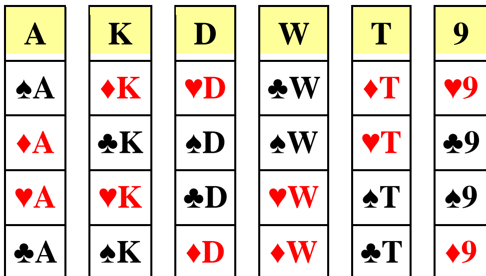
Załóżmy, że talia złożona z 24 kart potasowana jest następująco :



| ♥D | ♦K | ♣K | ♥9 | ♠D | ♣W | ♠A | ♥K | ♣9 | ♦T | ♦A | ♠K | ♥T | ♠W | ♥A | ♣D | ♦D | ♥W | ♠9 | ♣A | ♦9 | ♦W | ♠T | ♣T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

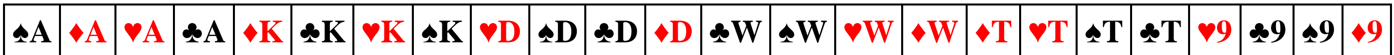
Chcemy je posortować szybko tak, aby najpierw występowały piki, później kiery, kara a na końcu trefle. W obrębie każdego koloru karty powinny być uporządkowane wg podanej powyżej kolejności. Postępujemy tak:

Najpierw kolejne karty układamy na 6 stosów wg figur (kolorem się chwilowo nie przejmujemy):



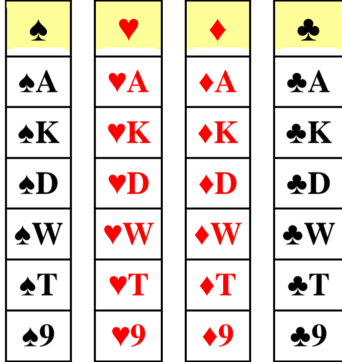
| A | K | D | W | T | 9 |
|---|---|---|---|---|---|
| ♠A | ♦K | ♥D | ♣W | ♦T | ♥9 |
| ♦A | ♣K | ♠D | ♠W | ♥T | ♣9 |
| ♥A | ♥K | ♣D | ♥W | ♠T | ♠9 |
| ♣A | ♠K | ♦D | ♦W | ♣T | ♦9 |

Poszczególne stosy łączymy ze sobą w talie kart - karty pobieramy z kupek w tej samej kolejności, w której były na nie wstawiane (u nas wstawianie rozpoczęliśmy od góry, zatem również od góry rozbieramy każdą z kupek):



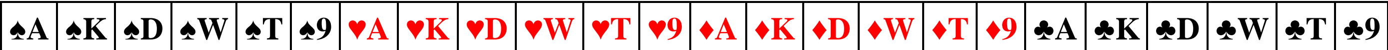
| ♠A | ♦A | ♥A | ♣A | ♦K | ♣K | ♥K | ♠K | ♥D | ♠D | ♣D | ♦D | ♣W | ♠W | ♥W | ♦W | ♦T | ♥T | ♠T | ♣T | ♥9 | ♣9 | ♠9 | ♦9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

W drugim kroku otrzymaną talie rozkładamy na 4 stosy wg kolorów:



| ♠ | ♥ | ♦ | ♣ |
|---|---|---|---|
| ♠A | ♥A | ♦A | ♣A |
| ♠K | ♥K | ♦K | ♣K |
| ♠D | ♥D | ♦D | ♣D |
| ♠W | ♥W | ♦W | ♣W |
| ♠T | ♥T | ♦T | ♣T |
| ♠9 | ♥9 | ♦9 | ♣9 |

Teraz wystarczy połączyć ze sobą otrzymane stosy w jedną talię 24 kart:



| ♠A | ♠K | ♠D | ♠W | ♠T | ♠9 | ♥A | ♥K | ♥D | ♥W | ♥T | ♥9 | ♦A | ♦K | ♦D | ♦W | ♦T | ♦9 | ♣A | ♣K | ♣D | ♣W | ♣T | ♣9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Talia kart jest posortowana.

***Lista kroków :***

Krok 1 : Przygotuj miejsce na tyle stosów, ile figur mogą mieć karty. Pierwszy stos będzie dla najstarszej karty, a ostatni dla najmłodszej.

Krok 2 : Rozdziel poszczególne karty na przygotowane wcześniej stosy wg figur - np. wszystkie asy idą do stosu asów, króle idą do stosu króli, itd.

Krok 3 : Złóż ze sobą karty w kolejnych stosach poczynając od stosu zawierającego najstarsze figury, a kończąc na stosie z najmłodszymi figurami.
Krok 4 : Przygotuj miejsce dla czterech stosów kolorów. Stosy powinny być ułożone w kolejności starszeństwa kolorów, np. piki, kiery, karo, trefle.
Krok 5 : Rozdziel karty na poszczególne stosy wg ich kolorów.
Krok 6 : Złóż ze sobą karty z kolejnych stosów poczynając od stosu zawierającego karty w najstarszym kolorze a kończąc na stosie z kartami w najmłodszym kolorze. Talia zostanie uporządkowana
Krok 7 : Zakończ algorytm

Należy zwrócić uwagę, iż przedstawiona metoda w ogóle nie porównuje elementów ze sobą. Elementy trafiają do odpowiednich stosów (są rozrzucane - stąd nazwa sortowanie rozrzutowe) na podstawie swojej wartości, a nie na podstawie ich relacji z innymi elementami zbioru. Dzięki takiemu podejściu zmieniona zostaje klasa czasowej złożoności obliczeniowej:

Pierwsza operacja - rozłożenie n elementów na m kupek ma klasę złożoności $\Theta(n)$.

Druga operacja - złączenie m kupek w n elementów ma klasę złożoności $\Theta(m)$.

Sumarycznie cały algorytm będzie posiadał klasę złożoności $\Theta$ (n + m). Jest to klasa liniowa, zatem sortowanie będzie bardzo szybkie. Co więcej, algorytm tego typu nie posiada przypadku pesymistycznego - czas sortowania każdego zbioru danych jest porównywalny. Mankament tego rozwiązania stanowi dodatkowe zapotrzebowanie na pamięć $\Theta(n)$ - musimy zarezerwować komórki na elementy przechowywane w stosach.

Przykładowy program

W programie wykorzystano opisany wyżej algorytm sortowania kart, który może stanowić część większego programu gry w karty. Należy zastrzec, iż problem sortowania kart rozwiązuje się o wiele prościej, jednakże tutaj chodzi o przedstawienie sposobu realizacji opisanego algorytmu. Zatem program ma raczej wartość dydaktyczną niż użytkową.

```
{ Sortowanie Kart }
{ -------------------------------------------------- }
{ (C)2005 mgr Jerzy Wałaszek }
{ I Liceum Ogólnokształcące  }
{ im. K. Brodzińskiego w Tarnowie }
{ -------------------------------------------------- }

Program skart;

Uses Crt;

{ Definicje typów danych }
type
  TKolor = (K_PIK,K_KIER,K_KARO,K_TREFL,K_PUSTY);
  TFigura = (F_A,F_K,F_D,F_W,F_10,F_9,F_8,F_7,F_6,F_5,F_4,F_3,F_2,F_0);
  TKarta = record
    Kolor : TKolor;
    Figura : TFigura;
  end;

{ Definicje zmiennych globalnych }

var
  talia : array[1..52] of TKarta;
```

```
  gracz : array[1..4,1..13] of TKarta;

{ Definicje procedur i funkcji }


{ -------------------------------------------------- }
{ Procedura inicjuje talię kart }
{ -------------------------------------------------- }
procedure Inicjuj_talie;
var
  i : integer;
  k : TKolor;
  f : TFigura;
begin
  k := K_PIK; f := F_A;
  for i := 1 to 52 do
  begin
    talia[i].Kolor := k; talia[i].Figura := f;
    inc(f);
    if f = F_0 then
    begin
      inc(k); f := F_A;
    end
  end;
end;

{ -------------------------------------------------- }
{ Procedura tasuje talię kart }
{ -------------------------------------------------- }
procedure Tasuj_talie;
var
  i,a,b : integer;
  x     : TKarta;
begin
  for i := 1 to 1000 do
  begin
    a := 1 + random(52); b := 1 + random(52);
    x := talia[a]; talia[a] := talia[b]; talia[b] := x;
  end;
end;

{ -------------------------------------------------- }
{ Procedura rozdaje karty poszczególnym graczom }
{ -------------------------------------------------- }
procedure Rozdaj_karty;
var
  i,j,k : integer;
begin
  k := 1;
  for i := 1 to 4 do
    for j := 1 to 13 do
    begin
      gracz[i,j] := talia[k];
      inc(k);
    end;
end;

{ -------------------------------------------------- }
{ Procedura sortuje karty gracza wg kolorów i figur }
{ -------------------------------------------------- }
procedure Sortuj_karty(g : integer);
var
  karty   : array[0..3,0..12] of TKarta;
```

```
  lfig    : array[TFigura] of integer;
  lkol    : array[TKolor] of integer;
  f   : TFigura;
  k   : TKolor;
  i,j : integer;
begin

 { Ustawiamy liczniki figur }

  for f := F_A to F_2 do lfig[f] := 0;

{ Przeglądamy rękę gracza i umieszczamy kolejne karty w tablicy }
{ figur wg figury }

  for i := 1 to 13 do
  begin
     f := gracz[g,i].Figura;
     karty[lfig[f],ord(f)] := gracz[g,i];
     inc(lfig[f]);
  end;

{ Przeglądamy tablicę figur pobierając z niej karty do ręki gracza }

  i := 1;
  for f := F_A to F_2 do
    for j := 0 to lfig[f] - 1 do
    begin
      gracz[g,i] := karty[j,ord(f)];
      inc(i);
    end;

{ Ustawiamy liczniki kolorów }

  for k := K_PIK to K_TREFL do lkol[k] := 0;

{ Przeglądamy rękę gracza i umieszczamy kolejne karty w tablicy }
{ kolorów wg koloru karty      }

  for i := 1 to 13 do
  begin
     k := gracz[g,i].Kolor;
     karty[ord(k),lkol[k]] := gracz[g,i];
     inc(lkol[k]);
  end;

{ Przeglądamy tablicę kolorów pobierając z niej karty do ręki gracza }

  i := 1;
  for k := K_PIK to K_TREFL do
    for j := 0 to lkol[k] - 1 do
    begin
      gracz[g,i] := karty[ord(k),j];
      inc(i);
    end;
end;

{ -------------------------------------------------   }
{ Procedura wyświetla karty gracza wg jego numeru     }
{ 1 : gracz u góry    (15,1)                          }
{ 2 : gracz po prawej (30,6)                          }
{ 3 : gracz u dołu    (15,11)                         }
{ 4 : gracz po lewej  (1,6)                           }
{ okna konsoli. Wydruk zajmuje 4 wiersze, w każdym do 15 znaków.     }
```

```
{ Figurę 10 wyświetlamy jako T - każda karta powinna zajmować jeden  }
{ znak, a zapis 10 wymaga dwóch znaków.                              }
{ ------------------------------------------------  }
procedure Wyswietl_karty(g : integer);
const
  kolory  : string[4]  = (#6#3#4#5);
  figury  : string[13] = ('AKDWT98765432');
  px      : array[1..4] of integer = (15,30,15,1);
  py      : array[1..4] of integer = (1,6,11,6);
var
  i : integer;
  k : TKolor;
begin
  for k := K_PIK to K_TREFL do
  begin
    gotoXY(px[g], py[g] + ord(k));
    write(kolory[1 + ord(k)],' ');
    for i := 1 to 13 do
      if gracz[g,i].Kolor = k then
        write(figury[1 + ord(gracz[g,i].Figura)]);
  end;
end;

{ ------------------------------------------------  }
{ Program główny                                    }
{ ------------------------------------------------  }

var
  i : integer;
begin
  Randomize;
  Inicjuj_talie;
  Tasuj_talie;
  Rozdaj_karty;
  for i := 1 to 4 do
  begin
    Sortuj_karty(i);
    Wyswietl_karty(i);
  end;
  gotoXY(1,16); writeln('Gotowe. Nacisnij klawisz Enter...');
  readln;
end.
```

### *8.11.2. Sortowanie kubełkowe.*

Opisany poniżej algorytm sortowania kubełkowego pozwala sortować zbiory liczb całkowitych - najlepiej o dużej ilości elementów, lecz o małym zakresie wartości. Zasada działania jest następująca:

1. Określamy zakres wartości, jakie mogą przyjmować elementy sortowanego zbioru. Niech $w_{min}$ oznacza najmniejszą wartość, a $w_{max}$ niech oznacza wartość największą.
2. Dla każdej możliwej wartości przygotowujemy kubełek-licznik, który będzie zliczał ilość wystąpień tej wartości w sortowanym zbiorze. Liczba liczników jest równa ($w_{max}$ - $w_{min}$ + 1). Każdy licznik jest początkowo ustawiony na wartość zero.
3. Przeglądamy kolejno elementy zbioru od pierwszego do ostatniego. Dla każdego elementu zbioru zwiększamy o jeden zawartość licznika o numerze równym wartości elementu. Na przykład, jeśli kolejny element zbioru ma wartość 3, to zwiększamy licznik o numerze 3. W efekcie po przeglądnięciu wszystkich elementów zbioru liczniki będą zawierały ilość wystąpień

każdej z możliwych wartości. Jeśli dany licznik zawiera 0, to wartość równa numerowi licznika w zbiorze nie występuje. Inaczej wartość ta występuje tyle razy, ile wynosi zawartość jej licznika.
4. Przeglądamy kolejne liczniki zapisując do zbioru wynikowego ich numery tyle razy, ile wynosi ich zawartość. Zbiór wyjściowy będzie posortowany.

Przykład : Posortujmy zbiór liczb { 2 6 4 3 8 7 2 5 7 9 3 5 2 6 }

Najpierw określamy zakres wartości elementów (w tym celu możemy na przykład wyszukać w zbiorze element najmniejszy i największy). U nas zakres wynosi:

$$w_{min} = 2,\ w_{max} = 9$$

Potrzebujemy zatem:

$$w_{max} - w_{min} + 1 = 9 - 2 + 1 = 8 \text{ liczników.}$$

Liczniki ponumerujemy zgodnie z wartościami, które będą zliczały:

[2] [3] [4] [5] [6] [7] [8] [9]

Na początku sortowania wszystkie liczniki mają stan zero:

[2:0] [3:0] [4:0] [5:0] [6:0] [7:0] [8:0] [9:0]

Teraz przeglądamy kolejne elementy zbioru zliczając ich wystąpienia w odpowiednich licznikach:

{ 2 6 4 3 8 7 2 5 7 9 3 5 2 6 }

[2:3] [3:2] [4:1] [5:2] [6:2] [7:2] [8:1] [9:1]

Zapis [2:3] oznacza, iż licznik numer 2 zawiera liczbę 3, a to z kolei oznacza, iż liczba 2 pojawiła się w zbiorze 3 razy. Przeglądamy kolejne liczniki począwszy od licznika o najmniejszym numerze (w przypadku sortowania malejącego przeglądanie rozpoczynamy od licznika o największym numerze) i zapisujemy do zbioru wynikowego tyle razy numer licznika, ile wynosi jego zawartość:

[2:3] [3:2] [4:1] [5:2] [6:2] [7:2] [8:1] [9:1]

{ 2 2 2 3 3 4 5 5 6 6 7 7 8 9 }

***Specyfikacja algorytmu.***

*Dane wejściowe :*

d [ ] - sortowany zbiór liczb całkowitych. Indeksy elementów rozpoczynają się od 1
n - liczba elementów w zbiorze, $n \in N$
$w_{min}$ - minimalna wartość elementu zbioru, $w_{min} \in C$
$w_{max}$ - maksymalna wartość elementu zbioru, $w_{max} \in C$

*Dane wyjściowe :*

d [ ] - Zbiór zawierający elementy posortowane

*Zmienne pomocnicze :*

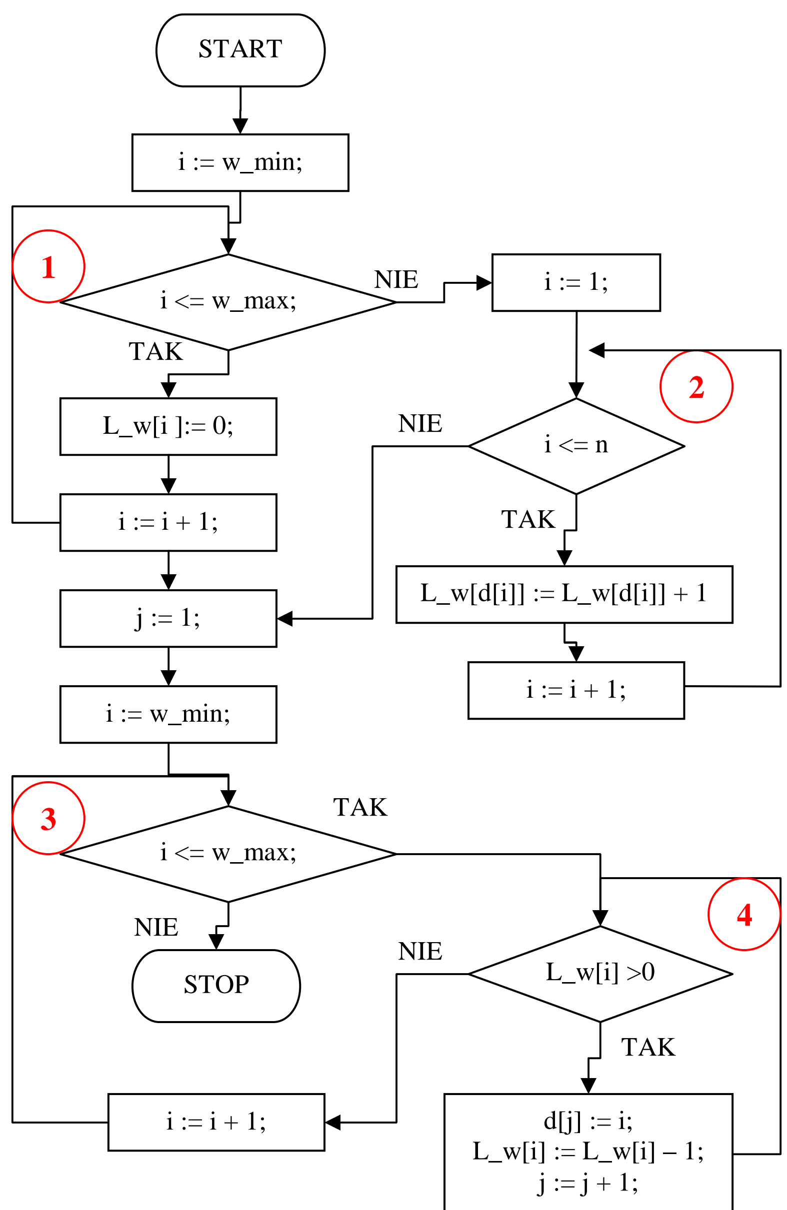
Lw [ ] - tablica liczników wartości o indeksach od $w_{min}$ do $w_{max}$. Każdy licznik przechowuje liczbę całkowitą

i, j - zmienne licznikowe pętli, $i, j \in C$

***Lista kroków :***

Krok 1 : Dla i = wmin , wmin + 1,...,wmax: wykonuj Lw[i] := 0
Krok 2 : Dla i = 1,2,...,n: wykonuj $L_w$[ d[i] ] := $L_w$ [d[i] ] + 1
Krok 3 : j := 1;
Krok 4 : Dla i = $w_{min}$ , $w_{min}$ + 1,...,$w_{max}$: wykonuj kroki 5...8
Krok 5 : Dopóki $L_w[i] > 0$ wykonuj kroki 6...8
Krok 6 : d[j] := i
Krok 7 : $L_w[i] := L_w[i] – 1$
Krok 8 : j := j + 1
Krok 9 : zakończ algorytm

***Schemat blokowy***




Algorytm realizujemy w czterech pętlach.
Pętla nr 1 zeruje kolejne liczniki L_w[ ].
W pętli nr 2 przeglądamy kolejne elementy zbioru od pierwszego do ostatniego. Dla każdego elementu zwiększamy licznik o numerze równym wartości elementu. Po zakończeniu tej pętli liczniki zawierają liczbę wystąpień poszczególnych wartości elementów w sortowanym zbiorze.
Zmienna j służy do umieszczania w zbiorze wyjściowym kolejnych elementów. Umieszczanie rozpoczniemy od początku zbioru, dlatego zmienne ta przyjmuje wartość 1.
Pętla nr 3 przegląda kolejne liczniki. Jeśli zawartość danego licznika jest większa od zera, to pętla nr 4 umieści w zbiorze wyjściowym odpowiednią ilość numerów licznika, które odpowiadają zliczonym wartościom elementów ze zbioru wejściowego.
Po zakończeniu pętli nr 3 elementy w zbiorze wyjściowym są posortowane. Kończymy algorytm.

Algorytm ma klasę czasowej złożoności obliczeniowej $\Theta(m + n)$, gdzie m oznacza ilość możliwych wartości, które mogą przyjmować elementy zbioru, a n to ilość sortowanych elementów. Jeśli m jest małe w porównaniu z n (sortujemy dużo elementów o małym zakresie wartości), to na czas sortowania będzie miała wpływ głównie ilość elementów n i klasa złożoności uprości się do postaci $\Theta(n)$. Dzieje się tak dlatego, iż przy równomiernym rozkładzie dużej ilości elementów o małym zakresie wartości liczniki będą równomiernie zapełnione (stan każdego licznika będzie dążył do $[^n/_m]$). Zatem algorytm wykona:

1. m operacji zerowania liczników - czas pomijalnie mały i przy dużym n nie wpływa istotnie na klasę algorytmu.
2. n operacji zwiększania liczników
3. n operacji przesłania numerów liczników do zbioru wynikowego - ilość pustych liczników będzie dążyła do zera.

W sytuacji odwrotnej, gdy sortujemy mało elementów o dużym zakresie wartości klasa złożoności zredukuje się z kolei do $\Theta(m)$.

Przykładowy program :

```
{ Sortowanie Kubełkowe 1 }
{ ------------------------------------------------   }
{ (C)2005 mgr Jerzy Wałaszek                         }
{ I Liceum Ogólnokształcące im. K. Brodzińskiego w Tarnowie }
{ ------------------------------------------------   }

program bucketsort1;

const WMIN = -99;
const WMAX = 99;
const N = 80;

var
  d   : array[1..N] of integer;
  lw  : array[WMIN..WMAX] of integer;
  i,j : integer;

begin
  writeln(' Sortowanie kubelkowe ');
  writeln('----------------------');
  writeln('(C)2005 Jerzy Walaszek');
  writeln;

{ tworzymy zbiór wejściowy do sortowania }

  randomize;
  for i := 1 to N do d[i] := WMIN + random(WMAX - WMIN + 1);

{ wyświetlamy zawartość zbioru przed sortowaniem }

  writeln('Przed sortowaniem:');
  writeln;
  for i := 1 to N do write(d[i]:4);
  writeln;

{ sortujemy }

{ najpierw zerujemy liczniki }
```

```
  for i := WMIN to WMAX do lw[i] := 0;

{ zliczamy w odpowiednich licznikach wystąpienia }
{ wartości elementów sortowanego zbioru }

  for i := 1 to N do inc(lw[d[i]]);

{ zapisujemy do zbioru wynikowego numery niezerowych liczników }
{ tyle razy, ile wynosi ich zawartość }

  j := 1;
  for i := WMIN to WMAX do
    while lw[i] > 0 do
    begin
      d[j] := i; inc(j); dec(lw[i]);
    end;

{ wyświetlamy zawartość zbioru po sortowaniu }

  writeln('Po sortowaniu:');
  writeln;
  for i := 1 to N do write(d[i]:4);

{ koniec }

  writeln;
  writeln('Gotowe. Nacisnij klawisz ENTER...');
  readln;
end.
```

### *8.11.3. Sortowanie przez zliczanie.*

Posortować rosnąco zbiór danych: { 6; 3; 6; 1; 4; 9; 0;1; 8 ;2; 6; 4; 9; 3; 7; 5; 9; 2; 7; 3; 2; 4; 1; 8; 7; 0; 8; 5; 8; 3; 6; 2; 5; 3 }

Przegladamy zbiór i wypisujemy wszystkie wartości w zbiorze : 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; a następnie przyporządkowujemy tym wartościom liczniki liczące ile razy dana wartość w zbiorze występuje. Oczywiście na początku wartość tych liczników jest równa zero (zaznaczono je kolorem zielonym)

**[0:0] [1:0] [2:0] [3:0] [4:0] [5:0] [6:0] [7:0] [8:0] [9:0]**

Przeglądamy kolejne elementy zbioru i zliczamy ich wystąpienia w odpowiednich licznikach. Np. element 6 powoduje zwiększenie o 1 licznika nr 6. Po wykonaniu tego obiegu w poszczególnych licznikach mamy ilość wystąpień każdej wartości. W naszym przykładzie otrzymamy:

**[0:2] [1:3] [2:4] [3:5] [4:3] [5:3] [6:4] [7:3] [8:4] [9:3]**

Teraz poczynając od drugiego licznika sumujemy zawartość licznika oraz jego poprzedników – wartość 0 wystąpiła 2 razy wartość 1 – 3 razy , wobec tego przy wartości 1 wpisujemy 2+3=5. Dla trzeciego licznika (wartość 2) zliczamy ilość wystąpień 0 (2 razy), 1 (3 razy) oraz 2 (4 razy) czyli przy wartości 2 wpisujemy 2+3+4=9. Analogicznie postępujemy tak dla wszystkich wartości otrzymując:

**[0:2] [1:5] [2:9] [3:14] [4:17] [5:20] [6:24] [7:27] [8:31] [9:34]**

W wyniku tej operacji w każdym liczniku otrzymaliśmy ilość wartości mniejszych lub równych numerowi licznika, które występują w zbiorze wejściowym. Na przykład:

[0:2] - w zbiorze wejściowym są dwie wartości 0
[1:5] - w zbiorze wejściowym jest pięć wartości mniejszych lub równych 1
[2:9] - w zbiorze wejściowym jest dziewięć wartość mniejszych lub równych 2, itd.

Zwróćmy uwagę, że stan licznika określa teraz ostatnią pozycję w zbiorze uporządkowanym, na której należy umieścić wartość równą numerowi licznika:

| **Wartość:** | **0** | **0** | **1** | **1** | **1** | **2** | **2** | **2** | **2** | **3** | **3** | **3** | **3** | **3** | **4** | **4** | **4** | **5** | **5** | **5** | **6** | **6** | **6** | **6** | **7** | **7** | **7** | **8** | **8** | **8** | **8** | **9** | **9** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Pozycja:** | 1 | **2** | 3 | 4 | **5** | 6 | 7 | 8 | **9** | 10 | 11 | 12 | 13 | **14** | 15 | 16 | **17** | 18 | 19 | **20** | 21 | 22 | 23 | **24** | 25 | 26 | **27** | 28 | 29 | 30 | **31** | 32 | 33 | **34** |

Przeglądamy jeszcze raz zbiór wejściowy idąc od ostatniego elementu do pierwszego (aby zachować stabilność sortowania). Każdy element umieszczamy w zbiorze wynikowym na pozycji równej zawartości licznika dla tego elementu. Po wykonaniu tej operacji licznik zmniejszamy o 1. Dzięki temu następna taka wartość trafi na wcześniejszą pozycję.

W naszym przykładzie rozpoczynamy od ostatniej liczby 3. Jej licznik ma zawartość [3:14]. Zatem liczbę 3 umieszczamy w zbiorze wynikowym na pozycji 14 i zmniejszamy o 1 stan licznika otrzymując [3:13]. Kolejna liczba 3 trafi teraz na pozycję 13, itd.

Dla liczby 5 stan licznika wynosi [5:20]. Umieszczamy ją zatem na 20 pozycji i licznik zmniejszamy o 1 otrzymując [5:19].

Postępujemy w ten sam sposób z pozostałymi elementami zbioru. W efekcie zbiór wynikowy będzie posortowany rosnąco:

{ 0 0 1 1 1 2 2 2 2 3 3 3 3 3 4 4 4 5 5 5 6 6 6 6 7 7 7 8 8 8 8 9 9 9 }

Przyjrzawszy się dokładnie algorytmowi sortowania przez zliczanie możesz zastanawiać się, dlaczego postępujemy w tak dziwny sposób? Przecież mając zliczone wystąpienia każdej wartości w licznikach, możemy je od razu przepisać do zbioru wyjściowego, jak zrobiliśmy w pierwszej wersji algorytmu sortowania kubełkowego. Miałbyś rację, gdyby chodziło jedynie o posortowanie liczb. Jest jednak inaczej.
Celem nie jest posortowanie jedynie samych wartości elementów. Sortowane wartości są zwykle tzw. kluczami, czyli wartościami skojarzonymi z elementami, które wyliczono na podstawie pewnego kryterium Sortując klucze chcemy posortować zawierające je elementy. Dlatego do zbioru wynikowego musimy przepisać całe elementy ze zbioru wejściowego, gdyż w praktyce klucze stanowią jedynie część (raczej małą) danych zawartych w elementach. Zatem algorytm sortowania przez zliczanie wyznacza docelowe pozycje elementów na podstawie reprezentujących je kluczy, które mogą się wielokrotnie powtarzać. Następnie elementy są umieszczane na właściwym miejscu w zbiorze wyjściowym.

***Specyfikacja algorytmu.***

*Dane wejściowe :*

d [ ] - zbiór elementów do posortowania. Każdy element posiada pole klucz, wg którego dokonuje się sortowania. Pole *klucz* jest liczbą całkowitą. Indeksy elementów rozpoczynają się od 1.

n - liczba elementów w zbiorze d [ ], $n \in N$

$k_{min}$ - minimalna wartość klucza, $k_{min} \in C$

$k_{max}$ - maksymalna wartość klucza, $k_{max} \in C$

*Dane wyjściowe :*

b [ ] - zbiór z posortowanymi elementami ze zbioru d[ ].
Indeksy elementów rozpoczynają się od 1

*Zmienne pomocnicze :*

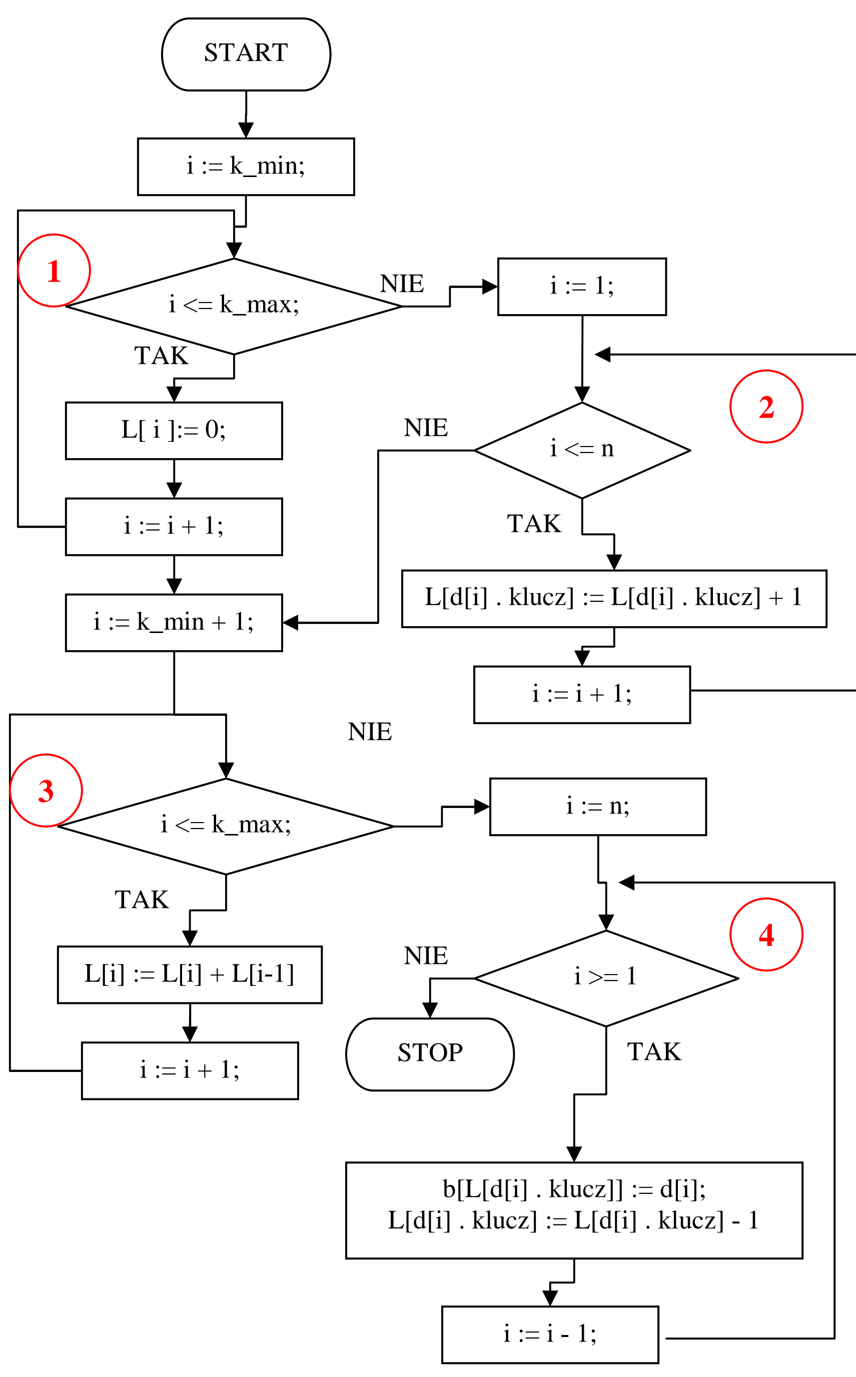
L [ ] - tablica liczników wartości kluczy. Elementy są liczbami całkowitymi. Indeksy przebiegają kolejne wartości od $k_{min}$ do $k_{max}$

i - zmienna licznikowa pętli, $i \in C$

***Lista kroków :***

Krok 1 : Dla $i = k_{min}, k_{min} + 1,...,k_{max}$: L[i] := 0;
Krok 2 : Dla i = 1,2,...,n: L[d[i] . klucz] := L[d[i] . klucz] + 1
Krok 3 : Dla $i = k_{min} + 1, k_{min} + 2,...,k_{max}$: L[i] := L[i] + L[i - 1]
Krok 4 : Dla i = n, n - 1,...,1: wykonuj kroki 5 i 6
Krok 5 : b[L[d[i].klucz]] := d[i]
Krok 6 : L[d[i].klucz] := L[d[i].klucz] - 1
Krok 7 : Zakończ algorytm

*Schemat blokowy*




Algorytm sortowania przez zliczanie zbudowany jest z kolejno następujących po sobie pętli iteracyjnych.

W pętli nr 1 przygotowujemy liczniki wystąpień poszczególnych kluczy. Ustawiamy je na 0.

W pętli nr 2 przeglądamy kolejne elementy zbioru zwiększając o 1 licznik o numerze równym wartości klucza w sortowanym elemencie zbioru. Po zakończeniu tej pętli w licznikach mamy ilość wystąpień poszczególnych kluczy.

W pętli numer 3 przekształcamy zliczone wartości wystąpień kluczy na ostatnie pozycje elementów z danym kluczem w zbiorze wyjściowym.

W pętli nr 4 ponownie przeglądamy zbiór wejściowy (idąc od końca do początku, aby zachować kolejność elementów równych - inaczej algorytm nie byłby stabilny) i przesyłamy elementy ze zbioru wejściowego do zbioru wyjściowego na pozycję o numerze zawartym w liczniku skojarzonym z kluczem elementu. Po przesłaniu licznik zmniejszamy o 1, aby kolejny element o tej samej wartości klucza trafił na poprzednią pozycję (idziemy wstecz, zatem kolejność elementów o tym samym kluczu zostanie zachowana). Po zakończeniu tej pętli dane w zbiorze wynikowym są posortowane rosnąco. Kończymy zatem algorytm.

Zwróć uwagę, iż algorytm sortowania przez zliczanie nie porównuje ze sobą żadnych elementów zbioru. Teoretycznie udowodniono, iż algorytmy sortujące, które porównują ze sobą elementy

zbioru nie mogą osiągać w przypadku ogólnym lepszej klasy złożoności obliczeniowej Θ(n log n). Ten algorytm nie porównuje elementów, zatem łamie tę granicę.
*Przykładowy program*

```
{ Sortowanie przez zliczanie                          }
{-----------------------------------------------------}
{ (C)2005 mgr Jerzy Wałaszek
{ I Liceum Ogólnokształcące im. K. Brodzińskiego w Tarnowie }
{-----------------------------------------------------}

const
  N    = 80;
  KMIN =  0;
  KMAX = 26;

{ Tutaj definiujemy typ elementu }

type
  TElement = record
    klucz : cardinal;
    wyraz : string[3];
  end;

{ Zmienne }

var
  d,b   : array[1..N] of TElement;
  L     : array[KMIN..KMAX] of cardinal;
  i,j,v : integer;
  s     : string;
begin
  writeln('  Sortowanie Przez Zliczanie  ');
  writeln('----------------------------');
  writeln('  (C)2005 mgr Jerzy Walaszek  ');
  writeln;
  writeln('Przed sortowaniem:');
  writeln;

{ Generujemy losowe elementy do sortowania oraz ich klucze }

  randomize;
  for i := 1 to N do
  begin
    s := '';
    for j := 1 to 3 do s := s + char(65 + random(3));
    d[i].wyraz := s;
    d[i].klucz := 0;
    v := 1;
    for j := 3 downto 1 do
    begin
      inc(d[i].klucz, v * (ord(d[i].wyraz[j]) - 65));
      v := 3 * v;
    end;
  end;

{ Wyświetlamy wygenerowane elementy }

  for i := 1 to N do write(d[i].wyraz:4);
  writeln;

{ Zerujemy liczniki }
```

```
  for i := KMIN to KMAX do L[i] := 0;

{ Zliczamy wystąpienia kluczy }

  for i := 1 to N do inc(L[d[i].klucz]);

{ Obliczamy pozycje końcowe elementów }

  for i := KMIN + 1 to KMAX do inc(L[i], L[i - 1]);

{ Przepisujemy elementy z d[ ] do b[ ] }

  for i := N downto 1 do
  begin
    b[L[d[i].klucz]] := d[i];
    dec(L[d[i].klucz]);
  end;

{ Wyświetlamy wyniki w b[ ] }

  writeln('Po sortowaniu:');
  writeln;
  for i := 1 to N do write(b[i].wyraz:4);
  writeln;
  writeln('Koniec. Nacisnij klawisz Enter...');
  readln;
end.
```

## 8.12. Podsumowanie.

| Nazwa algorytmu sortującego | Klasa złożoności | | | Stabilność | Sortowanie w miejscu | Zalecane? |
|---|---|---|---|---|---|---|
| | optymistyczna | typowa | pesymistyczna | | | |
| Zwariowane | $\Theta(1)$ | $\Theta(n*n!)$ | $\Theta(\infty)$ | NIE | TAK | NIE!!! |
| Naiwne | $\Theta(n) ... \Theta(n^2)$ | $\Theta(n^3)$ | $\Theta(n^3)$ | TAK | TAK | NIE |
| Bąbelkowe wersja 1 | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | TAK | TAK | NIE |
| Bąbelkowe wersja 2 | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | TAK | TAK | TAK |
| Przez wybór | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | NIE | TAK | TAK/NIE |
| Przez wstawianie | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | TAK | TAK | TAK |
| Metodą Shella | $\Theta(n^{1,14})$ | $\Theta(n^{1,15})$ | $\Theta(n^{1,15})$ | NIE | TAK | TAK |
| Przez łączenie | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | TAK | NIE | TAK |
| Przez kopcowanie | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | NIE | TAK | TAK |
| Szybkie | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n^2)$ | NIE | TAK | TAK |
| Kubełkowe wersja I | $\Theta(m + n)$ | $\Theta(m + n)$ | $\Theta(m + n)$ | NIE | NIE | TAK/NIE |
| Radix Sort | $\Theta(n \log n)$ | $\Theta(n \log n)$ | $\Theta(n \log n)$ | TAK | NIE | TAK |

Najszybsze algorytmy sortujące to algorytmy sortowania dystrybucyjnego. Jednakże szybkość ich działania jest okupiona dużym zapotrzebowaniem na pamięć. Algorytmy te mają liniową klasę czasowej złożoności obliczeniowej.

Z algorytmów sortujących w miejscu najszybszym w typowych warunkach jest algorytm sortowania szybkiego. Posiada liniowo logarytmiczną klasę czasowej złożoności obliczeniowej. Jednakże dla niekorzystnych danych może się degradować do klasy kwadratowej.
Wolniejszym (około dwa razy) algorytmem sortowania klasy liniowo logarytmicznej jest algorytm sortowania stogowego. Algorytm ten nie degraduje się do niższej klasy złożoności obliczeniowej i może być alternatywą dla algorytmu sortowania szybkiego.
Bardzo obiecujące wyniki otrzymaliśmy dla algorytmu sortowania metodą Shella.

W klasie kwadratowej złożoności obliczeniowej zalecany jest do stosowania algorytm sortowania przez wstawianie. Jest bardzo prosty w implementacji i jednocześnie wystarczająco szybki. Dla zbiorów w znacznym stopniu uporządkowanych wykazuje liniową klasę złożoności obliczeniowej. Dlatego nadaje się np. do sortowania zbioru uporządkowanego, do którego dodajemy nowy element - zbiór będzie posortowany szybciej niż przez algorytm sortowania szybkiego. Ten wniosek jasno pokazuje, iż nie ma uniwersalnych algorytmów sortujących.
Algorytmów sortowania bąbelkowego raczej należy unikać.

# 9. Binarne kodowanie liczb.

Zadanie 1.
Proszę wpisać poniższy kod, skompilować program i uruchomić go.

```
Program Zadanie_1A;
Var a : Byte;
Begin
        a := 255;
        a := a + 1;
        Writeln(a);
        Writeln;
        Writeln('Nacisnij ENTER...');
        Readln;
End.

Program Zadanie_1B;
Var a : Real;
Begin
        a := 0.1;
        Writeln (10*a);
        If 10 * a = 1 Then Writeln('Mam') Else Writeln('Nie mam');
        Writeln;
        Writeln('Naciśnij ENTER...');
        Readln;
End.


Program Zadanie_1C;
Var a : Single;
    i : Integer;
Begin
        a := 0;
        For i := 1 to 20000000 Do a := a + 1;
        Writeln(a:0:0);
        Writeln('Naciśnij ENTER...');
        Readln;
End.
```

Powyższe zadania (programy) pokazują zadziwiające wyniki. Dlaczego ?

## *9.1. System dziesiętny.*

Podstawę systemu dziesiętnego tworzy liczba 10. Jest to specjalna wartość charakteryzująca system pozycyjny, od której bierze on swoją nazwę: podstawa 10 - system dziesiętny.
Zapis liczby tworzymy za pomocą cyfr, czyli umownych znaków o przypisanych wartościach od 0 do 9. Ilość cyfr jest zawsze równa podstawie systemu, czyli w systemie dziesiętnym będzie ich dziesięć. Największa cyfra jest o 1 mniejsza od podstawy (9 = 10 - 1).
Cyfry umieszczamy na kolejnych pozycjach. Każda pozycja posiada swoją wartość, którą nazywamy wagą pozycji. Wagi pozycji są kolejnymi potęgami podstawy systemu, czyli w systemie dziesiętnym są to kolejne potęgi liczby 10:

| **wagi** | 1000 $10^3$ | 100 $10^2$ | 10 $10^1$ | 1 $10^0$ |
|---|---|---|---|---|
| **cyfry** | **7** | **5** | **8** | **2** |
| **pozycje** | 3 | 2 | 1 | 0 |

Cyfra na danej pozycji określa ile razy należy wziąć wagę tej pozycji przy wyznaczaniu wartości całej liczby. Zatem w naszym przykładzie waga 1000 występuje **7** razy, waga 100 - **5** razy, waga 10 - **8** razy i waga 1 - **2** razy. Wartość liczby otrzymujemy sumując iloczyny cyfr przez wagi pozycji, na których cyfry te występują (czyli ilości tych wag):
**7** *1000 + **5***100 + **8** *10 + **2** *1
Jeśli pozycje ponumerujemy kolejno od 0 poczynając od prawej strony zapisu liczby, to waga pozycji i-tej będzie i-tą potęgą podstawy systemu. Np. pozycja nr 3 ma wagę $10^3$, czyli 1000, a pozycja nr 2 ma wagę $10^2$, czyli 100.

Analogicznie możemy zapisać liczbę w każdym innym systemie , np. dwójkowym (binarnym), ósemkowym czy szestnastowym – podstawą będzie wtedy odpowiednio liczba 2, 8 lub 16.

**Zbiór podstawowych cech dowolnego systemu pozycyjnego o podstawie p**

1. System pozycyjny charakteryzuje liczba zwana podstawą systemu pozycyjnego.
2. Do zapisu liczby służą cyfry.
3. Cyfr jest zawsze tyle, ile wynosi podstawa systemu: 0,1,2,...,(p-1)
4. Cyfry ustawiamy na kolejnych pozycjach.
5. Pozycje numerujemy od 0 poczynając od strony prawej zapisu.
6. Każda pozycja posiada swoją wagę.
7. Waga jest równa podstawie systemu podniesionej do potęgi o wartości numeru pozycji.
8. Cyfry określają ile razy waga danej pozycji uczestniczy w wartości liczby
9. Wartość liczby obliczamy sumując iloczyny cyfr przez wagi ich pozycji

Wartość dziesiętna liczby w systemie pozycyjnym o podstawie p $C_{n-1}C_{n-2}...C_2C_1C_0$ ma wartość $C_{n-1} p^{n-1} + C_{n-2} p^{n-2} + ... + C_2 p^2 + C_1 p^1 + C_0 p^0$
gdzie:

C - cyfra danego systemu o podstawie p
$C_i$ - cyfra na i-tej pozycji, i = 0,1,2,...,n-1
n - ilość cyfr w zapisie liczby
p - podstawa systemu pozycyjnego

*Przykład : Obliczyć wartość (w systemie dziesiętnym) liczby 53214 $_{(6)}$ (zapisanej w systemie szóstkowym)*

1) Ponad cyframi wypisujemy wagi pozycji, pamiętając, iż pozycje numerujemy od 0 z prawa na lewo. Wagi są kolejnymi potęgami podstawy systemu. Waga danej pozycji jest zawsze równa podstawie podniesionej do potęgi o wartości numeru pozycji.

| wartość wagi | $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ |
|---|---|---|---|---|---|
| wartość cyfry | **5** | **3** | **2** | **1** | **4** |
| numer pozycji | 4 | 3 | 2 | 1 | 0 |

2) Tworzymy sumę iloczynów cyfr przez wagi ich pozycji

| | | | | | | |
|---|---|---|---|---|---|---|
| wartość wagi | $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ | |
| wartość cyfry | **5** | **3** | **2** | **1** | **4** | = **5*** $6^4$ + **3*** $6^3$ + **2*** $6^2$ + **1*** $6^1$ + **4*** $6^0$ |
| numer pozycji | 4 | 3 | 2 | 1 | 0 | |

3) Wyliczamy wagi kolejnych pozycji

| | | | | | | |
|---|---|---|---|---|---|---|
| wartość wagi | $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ | |
| wartość cyfry | **5** | **3** | **2** | **1** | **4** | = **5*** 1296 + **3*** 216 + **2*** 36 + **1*** 6 + **4*** 1 |
| numer pozycji | 4 | 3 | 2 | 1 | 0 | |

4) Obliczamy iloczyny cyfr przez wagi pozycji

| | | | | | | |
|---|---|---|---|---|---|---|
| wartość wagi | $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ | |
| wartość cyfry | **5** | **3** | **2** | **1** | **4** | = 6480 + 648 + 72 + 6 + 4 |
| numer pozycji | 4 | 3 | 2 | 1 | 0 | |

5) Sumujemy iloczyny i otrzymujemy wartość liczby

| | | | | | | |
|---|---|---|---|---|---|---|
| wartość wagi | $6^4$ | $6^3$ | $6^2$ | $6^1$ | $6^0$ | |
| wartość cyfry | **5** | **3** | **2** | **1** | **4** | = **7210** |
| numer pozycji | 4 | 3 | 2 | 1 | 0 | |

I ostatecznie piszemy $53214_{(6)} = 7210_{(10)}$. Jeśli operujemy liczbami zapisanymi w różnych systemach pozycyjnych, to w celu uniknięcia niejednoznaczności liczbę opatrujemy indeksem dolnym, w którym umieszczamy wartość podstawy systemu zapisu danej liczby. Powyższa równość oznacza, iż zapis szóstkowy i dziesiętny oznacza tę samą liczbę.

**Algorytm obliczania wartości liczby pozycyjnej.**

*Specyfikacja problemu algorytmicznego.*

*Dane wejściowe :*

p - podstawa systemu pozycyjnego zapisu liczby $p \in N$ , $p \in$ <2,3, n>

s - tekst przedstawiający liczbę

*Dane wyjściowe :*

L - liczba będąca wartością liczby o podstawie p i zapisanej w postaci ciągu znaków s

*Zmienne pomocnicze :*

w - wagi kolejnych pozycji, $w \in N$

x - przechowuje wartość znaku s[i];

***Lista kroków***

Krok 1 : czytaj p oraz s
Krok 2 : L := 0; w:=1;
Krok 3 : Dla i = długość (s) , dlugosc(s) –1, .. 2, 1 wykonuj kroki 4...7
Krok 4 : Przypisz x liczbę s[i]
Krok 5 : Jeśli i = dlugość to L := x;
Krok 6 : w := w*p;
Krok 7 : L := L + x*w;
Krok 8 : Wyswietl wartość L i zakończ algorytm

## *9.2. Schemat Hornera.*

Obliczanie wartości wielomianu

Rozważmy wielomiany :

$w(x) = a_0x^2 + a_1x + a_2 = (a_0x + a_1)\ x + a_2$
W tym wielomianie (drugiego stopnia) są wykonywane 2 mnożenia i 2 dodawania

$w(x) = a_0x^3 + a_1x^2 + a_2x + a_3 = ((a_0x + a_1)\ x + a_2)\ x + a_3$
W wielomianie 3 stopnia wykonamy 3 mnożenia i 3 dodawania

Ogólnie :
$w(x) = a_0x^n + a_1x^{n-1} + ....... + a_{n-1}x + a_n = (\ a_0x^{n-1} + a_1x^{n-2} + ....... + a_{n-1})\ x + a_n =$
$((\ a_0x^{n-2} + a_1x^{n-3} + ....... + a_{n-2}))\ x + a_{n-1})\ x + a_n = ......... =$
$((.......(\ a_0x + a_1)\ x + a_2\ )\ x + ....... + a_{n-2}))\ x + a_{n-1})\ x + a_n$

Ilość mnożeń i dodawań w wielomianie stopnia n wynosi n dla każdego z działań

**Dane** : n – liczba całkowita nieujemna – stopień wielomianu
$a_0$ , $a_1$ , $a_2$ , ....., $a_n$ - współczynniki wielomianu , ilość - n+1
z – wartość argumentu
y – zmienna pomocnicza
**Wynik** : Wartość wielomianu stopnia n dla wartości argumentu x = z

Aby obliczyć wartość wielomianu dla wartości argumentu z należy postępować następująco :

1) $y := a_0$
2) $y := yz + a_1$
3) $y := yz + a_2$
............
n) $y := yz + a_{n-1}$
n+1) $y := yz + a_n$

Oczywiście pamiętajmy , że obliczoną wartość y np. z wiersza 1) podstawiamy do wiersza 2 itp.

Wszystkie wiersze za wyjątkiem pierwszego, możemy zapisać w postaci
$y := a_0$
$y := yz + a_i$ gdzie i = 1,2,3, ..... ,n

Taki sposób obliczania wartości wielomianu nazywa się **schematem Hornera**. Jest to najszybszy sposób obliczania wartości wielomianu.

Zastosowanie schematu Hornera :

1) obliczanie wartości dziesiętnej liczb danych innym systemie pozycyjnym
2) szybkie obliczanie wartości potęgi wykorzystywanego w szyfrowaniu z kluczem jawnym
3) jednoczesne obliczanie wartości wielomianu i jego pochodnej

## *9.3. Schemat Hornera jako sposób obliczania wartości liczby.*

W rozdziale „System dziesiętny” został podany algorytm obliczania wartości liczby na podstawie wzoru:

$$C_{n-1}C_{n-2}...C_2C_1C_0 = C_{n-1}\, p^{n-1} + C_{n-2}\, p^{n-2} + ... + C_2\, p^2 + C_1\, p^1 + C_0\, p^0$$

W zastosowaniach informatycznych korzysta się z innego rozwiązania, zwanego schematem Hornera. Właściwie schemat ten ma zastosowanie przy wyznaczaniu wartości wielomianu, lecz jeśli przyjrzymy się dokładnie powyższemu wzorowi, to na pewno zauważymy podobieństwo do wzoru na wartość wielomianu:

$$W(x) = a_{n-1}\, x^{n-1} + a_{n-2}\, x^{n-2} + ... + a_2\, x^2 + a_1\, x + a_0$$

Współczynniki $a_i$ dla $i = 0,1,2,...,n-1$ odpowiadają wartościom cyfr C. Natomiast kolejne potęgi zmiennej x to oczywiście potęgi podstawy p. Schemat Hornera wyznaczymy dla 5-cio cyfrowej liczby (dla n-cyfrowej zasada jest identyczna). Liczba zapisana jest w systemie pozycyjnym o podstawie p ciągiem cyfr $C_4C_3C_2C_1C_0$ i ma wartość:

$$L = C_4\, p^4 + C_3\, p^3 + C_2\, p^2 + C_1\, p^1 + C_0\, p^0$$

Ponieważ $p_1 = p$ oraz $p_0 = 1$, powyższy wzór można nieco uprościć i zapisać go w postaci:

$$L = C_4\, p^4 + C_3\, p^3 + C_2\, p^2 + C_1\, p + C_0$$

Wyprowadzamy przed nawias wspólny czynnik p:

$$L = p\, (C_4\, p^3 + C_3\, p^2 + C_2\, p + C_1) + C_0$$

Zwróć uwagę, iż wyrażenie w nawiasie ma niższy stopień. Znów wyprowadzamy przed nawias wspólny czynnik p.

$$L = p\, (p\, (C_4\, p^2 + C_3\, p + C_2) + C_1) + C_0$$

I jeszcze raz:

$$L = p\, (p\, (p\, (C_4\, p + C_3) + C_2) + C_1) + C_0$$

I po raz ostatni:

$$L = p\, (p\, (p\, (p\, (C_4) + C_3) + C_2) + C_1) + C_0$$

Ze względu na przemienność operacji mnożenia otrzymany wzór możemy zapisać w postaci:

$$L = ((((C_4)\, p + C_3)\, p + C_2)\, p + C_1)\, p + C_0$$

Teraz wartość liczby obliczamy wyliczając wartości wyrażeń w kolejnych nawiasach:

$L_0 = C_4$ - wartość początkowa

$L_1 = L_0\, p + C_3 = C_4\, p + C_3$

$L_2 = L_1\, p + C_2 = (C_4\, p + C_3)\, p + C_2 = C_4\, p^2 + C_3\, p + C_2$

$L_3 = L_2\, p + C_1 = (C_4\, p^2 + C_3\, p + C_2)\, p + C_1 = C_4\, p^3 + C_3\, p^2 + C_2\, p + C_1$

$L_4 = L_3\, p + C_0 = (C_4\, p^3 + C_3\, p^2 + C_2\, p + C_1)\, p + C_0 = C_4\, p^4 + C_3\, p^3 + C_2\, p^2 + C_1\, p + C_0$

Zwróćmy uwagę na sposób wyliczania wartości liczby. Wyraźnie widoczny jest pewien schemat postępowania. Najpierw za wartość liczby przyjmujemy $C_4$. Następnie do wyczerpania pozostałych cyfr wykonujemy te same obliczenia: nową wartość otrzymujemy mnożąc poprzednią wartość

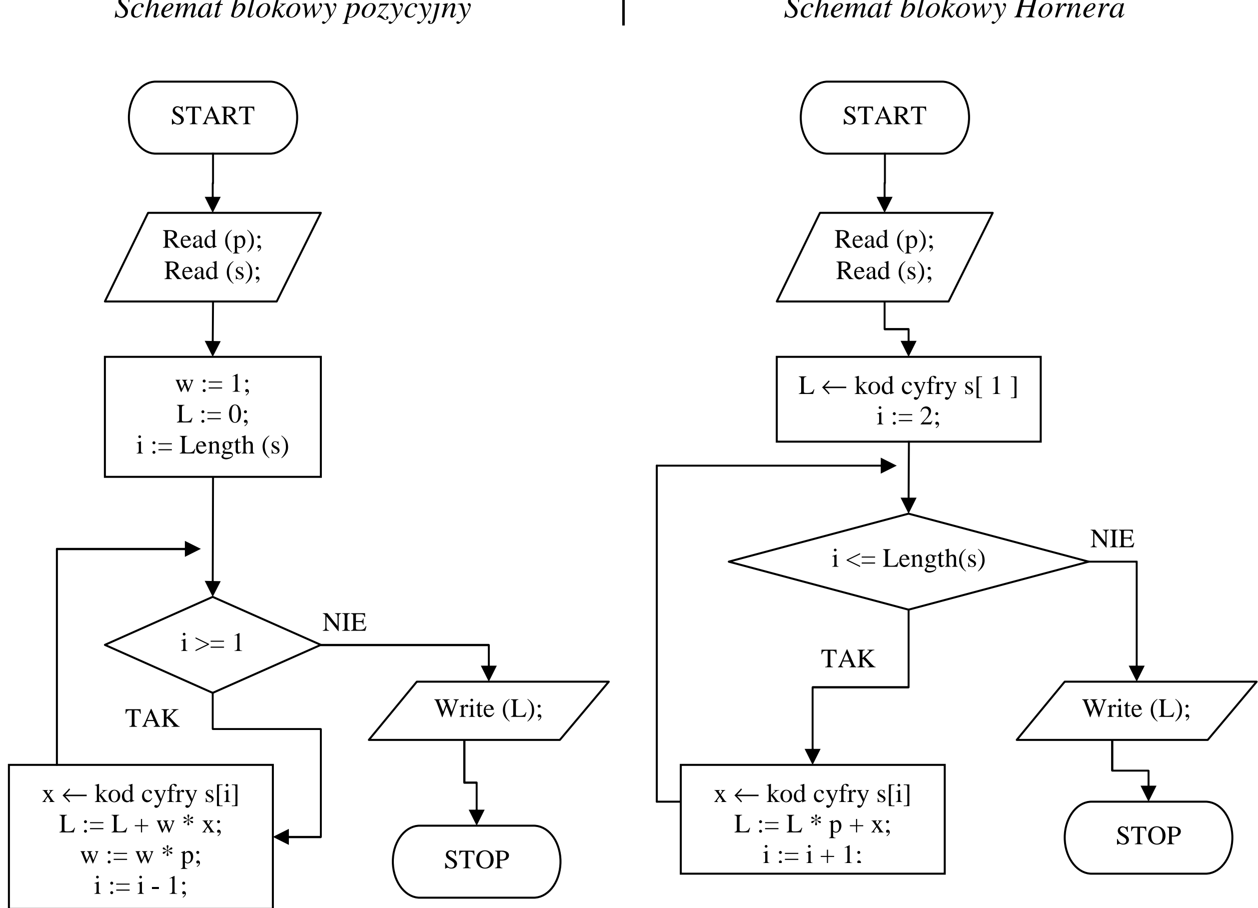
przez podstawę systemu i dodając kolejną cyfrę. Rachunki kończymy po dodaniu ostatniej cyfry zapisu liczby.

*Przykład Obliczyć za pomocą schematu Hornera wartość liczby piątkowej $4223213_{(5)}$.*
$L_0 = 4$
$L_1 = 4 * 5 + 2 = 22$
$L_2 = 22 * 5 + 2 = 112$
$L_3 = 112 * 5 + 3 = 563$
$L_4 = 563 * 5 + 2 = 2817$
$L_5 = 2817 * 5 + 1 = 14086$
$L_6 = 14086 * 5 + 3 = 70433$ - koniec, ponieważ wyczerpaliśmy wszystkie cyfry
$4223213_{(5)} = 70433_{(10)}$.

Po co to wszystko jest potrzebne? Po pierwsze oszczędność w mnożeniu. Sprawdźmy. Dla pięciu cyfr musimy wykonać następujące rachunki przy zastosowaniu standardowego wzoru:
$L = C_4\ p\ p\ p\ p + C_3\ p\ p\ p + C_2\ p\ p + C_1\ p + C_0$

Daje to w sumie 10 mnożeń i 4 dodawania. Ten sam rachunek schematem Hornera prowadzi do wykonania 4 mnożeń i 4 dodawań. Mniej mnożeń oznacza większą efektywność algorytmu Hornera ponieważ mnożenie zajmuje procesorowi komputera więcej czasu od dodawania. Drugą zaletą jest sposób przetwarzania cyfr. Bierzemy je kolejno jedna po drugiej z ciągu wejściowego aż do napotkania końca zapisu. Ponieważ taka kolejność cyfr jest zwykle zgodna z kolejnością ich przechowywania w łańcuchu tekstowym, zatem sposób ten daje nam kolejne przyspieszenie i uproszczenie działania algorytmu (w poprzednim algorytmie cyfry przetwarzaliśmy w kierunku odwrotnym poczynając od ostatniej w zapisie).

*Schemat blokowy pozycyjny* | *Schemat blokowy Hornera*



*Przykładowy program (schemat Hornera)*

```
Program Zamiana_Systemow_Liczbowych; {schemat Hornera }

Uses Crt;

Var s        : String;
   L,x,p,i   : LongInt;

           { ------------------------------------------------ }
           Function Przypisz_Wartosc (znak : Char) : Integer;
           Var code : Integer;
           Begin
             znak := UpCase(znak);
             Case znak of
                 'A' : x := 10;
                 'B' : x := 11;
                 'C' : x := 12;
                 'D' : x := 13;
                 'E' : x := 14;
                 'F' : x := 15;
                 Else Val (znak,x,code);
                  End;
             Przypisz_Wartosc := x;
           End;
           { ------------------------------------------------ }

Begin
ClrScr;
    Writeln ('---------------------------------------------------------------------------');
    Writeln ('    Program zamieniajacy liczbe podana w dowolnym systemie    ');
    Writeln ('   pozycyjnym ( 2 ... 16)  na wartosc w systemie dziesietnym  ');
    Writeln ('---------------------------------------------------------------------------');
    Writeln;
    Write (' Podaj podstawe systemu  od 2 do 16 : ');
    Readln (p);
    Write (' Podaj liczbe, ktorej wartosc chcesz znac w systemie dziesietnym :  ');
    Readln (s);
    i := 2;
    L := Przypisz_Wartosc (s[1]);
    For i := 2 to Length(s) Do
    Begin
       x := Przypisz_Wartosc (s[i]);
       L := L * p + x;
    End;
    Writeln;
    Writeln ('  Liczba ',s,'(',p,')  = ',L,'(10)');
    Writeln;
    Writeln (' Nacisnij dowolny klawisz ');
    Repeat Until KeyPressed;
End.
```

## *9.4. Przeliczanie liczb systemu dziesiętnego na dowolny system*

Zadanie : Przedstawić liczbę $4321_{(10)}$ w systemie trójkowym.
Rozwiązanie :
Wartość liczby L wyraża się wzorem $L = C_{n-1}\,p^{n-1} + C_{n-2}\,p^{n-2} + ... + C_2\,p^2 + C_1\,p^1 + C_0\,p^0$
gdzie : p – podstawa systemu, C – wartość cyfry

Tak więc problemem będzie znalezienie wartości cyfr $C_{n-1}$ , $C_{n-2}$ , ..... , $C_2$ , $C_1$ , $C_0$ w systemie dziesiętnym. Użyjemy do tego celu dzielenia całkowitoliczbowego **DIV** oraz reszty z dzielenia całkowitoliczbowego **MOD**

Czyli :

| | | | |
|---|---|---|---|
| 4321 | div 3 = | 1440 | reszta 1 |
| 1440 | div 3 = | 480 | reszta 0 |
| 480 | div 3 = | 160 | reszta 0 |
| 160 | div 3 = | 53 | reszta 1 |
| 53 | div 3 = | 17 | reszta 2 |
| 17 | div 3 = | 5 | reszta 2 |
| 5 | div 3 = | 1 | reszta 2 |
| 1 | div 3 = | 0 | reszta 1 |

Wpisujemy reszty w kolejności od dołu i otrzymujemy, że $4321_{(10)} = 12221001_{(3)}$

Obliczyć wartość $4321_{(10)}$ w systemie ósemkowym

| | | | |
|---|---|---|---|
| 4321 | div 8 = | 540 | reszta 1 |
| 540 | div 8 = | 67 | reszta 4 |
| 67 | div 8 = | 8 | reszta 3 |
| 8 | div 8 = | 1 | reszta 0 |
| 1 | div 8 = | 0 | reszta 1 |

Wpisujemy reszty w kolejności od dołu i otrzymujemy, że $4321_{(10)} = 10341_{(8)}$

Obliczyć wartość $4321_{(10)}$ w systemie szestnastkowym

| | | | |
|---|---|---|---|
| 4321 | div 16 = | 270 | reszta 1 |
| 270 | div 16 = | 16 | reszta 14 |
| 16 | div 16 = | 1 | reszta 0 |
| 1 | div 16 = | 0 | reszta 1 |

Wpisujemy reszty w kolejności od dołu pamiętając, że cyfry powyżej 9 zastępujemy literami czyli tu 14=E i otrzymujemy, że $4321_{(10)} = 10E1_{(16)}$

***Algorytm wyznaczania wartości liczby w systemie dziesiętnym na dowolny system***

*Wejście:* W - wartość liczby ,
p – podstawa liczby wyjściowej

*Wyjście:* S - ciąg znaków reprezentujących wartość liczby $W_{(10)}$ w systemie p

*Zmienna pomocnicze :*

r - reszta z dzielenia całkowitoliczbowego

krok 1 : r := W mod p; W := W div p;
krok 2 : Zamień r na znak i dodaj do łancucha S
krok 3 : Jeśli W > 0, wróć do kroku 1
krok 4 : Wyprowadź łańcuch S (reszty z dzielenia) w kolejności odwrotnej
krok 5 : Koniec

Zadanie : Napisać program przeliczający wartość liczby w systemie dziesiętnym na liczbę w dowolnym systemie.

## *9.5. Liczby zmiennoprzecinkowe.*

Z liczbami zmiennoprzecinkowymi (ang. floating point numbers) każdy się spotkał już na przykład na lekcjach fizyki. Zapis bardzo dużych lub bardzo małych liczb w normalnej notacji pozycyjnej jest niewygodny, gdyż wymaga sporej ilości cyfr. Dlatego takie liczby zapisuje się w postaci zmiennoprzecinkowej czyli na przykład :

$2{,}998 \times 10^{8}$ (prędkość światła w próżni w m/s ☺ ) lub $1{,}602 \times 10^{-19}$ (ładunek elektronu w C ☺ )

Zapis składa się z trzech liczb:

**m** - mantysy, np. 2,998
**p** - podstawy systemu, np. 10
**c** - cechy, np. 8

Wartość liczby zmiennoprzecinkowej obliczamy zgodnie ze wzorem:

$$\mathbf{L = m \times p^{C}}$$

Oczywiście wzór ten pozwala obliczyć wartość liczby zmiennoprzecinkowej zapisanej w dowolnym systemie pozycyjnym, a nie tylko dziesiętnym.

*Przykład : Obliczyć wartość zmiennoprzecinkowej liczby trójkowej* $2{,}21 \times 10^{21}{}_{(3)}$.

Najpierw obliczamy wartości (patrz – system dziesiętny) poszczególnych składników liczby zmiennoprzecinkowej pamiętając, iż każdy z nich jest zapisany w systemie trójkowym:

$\mathbf{m} = 2{,}21_{(3)} = 2 \times 3^{0} + 2 \times 3^{-1} + 1 \times 3^{-2} = 2 \times 1 + 2 \times {}^{1}/_{3} + 1 \times {}^{1}/_{9} = 2 + {}^{2}/_{3} + {}^{1}/_{9} = \mathbf{2\,{}^{7}/_{9}}$

$\mathbf{p} = 10_{(3)} = 1 \times 3^{1} + 0 \times 3^{0} = \mathbf{3}$

$\mathbf{c} = 21_{(3)} = 2 \times 3^{1} + 1 \times 3^{0} = 2 \times 3 + 1 \times 1 = 6 + 1 = \mathbf{7}$

Teraz wykorzystujemy wzór $L = m \times p^{C} = 2\,{}^{7}/_{9} \times 3^{7} = {}^{25}/_{9} \times 3^{7} = 25 \times 3^{5} = 25 \times 243 = \mathbf{6075}$

*Przykład : Obliczyć wartość zmiennoprzecinkowej liczby szesnastkowej* $\mathrm{A{,}CB} \times 10^{\mathrm{D}}$.

$m = \mathrm{A{,}CB}_{(16)} = \mathrm{A} \times 16^{0} + \mathrm{C} \times 16^{-1} + \mathrm{B} \times 16^{-2}$
$m = \mathrm{A{,}CB}_{(16)} = 10 \times 16^{0} + 12 \times 16^{-1} + 11 \times 16^{-2}$
$m = \mathrm{A{,}CB}_{(16)} = 10 \times 1 + 12 \times {}^{1}/_{16} + 11 \times {}^{1}/_{256} = 10 + {}^{12}/_{16} + {}^{11}/_{256} = 10 + {}^{192}/_{256} + {}^{11}/_{256}$
$m = \mathrm{A{,}CB}_{(16)} = 10\,{}^{203}/_{256}$

$p = 10_{(16)} = 1 \times 16^{1} + 0 \times 16^{0} = 16$
$c = \mathrm{D}_{(16)} = \mathrm{D} \times 16^{0} = 13 \times 16^{0} = 13$

$L = m \times p^c = 10\ ^{203}/_{256} \times 16^{13} = ^{2763}/_{256} \times 16^{13} = 2763 \times 16^{11} = 2763 \times 17592186044416$
$L = 48607210040721408$

Ponieważ liczbę zmiennoprzecinkową można zapisywać w różny sposób, przyjęto tzw. postać znormalizowaną.

$$325 \times 10^{20} = 32{,}5 \times 10^{21} = 3{,}25 \times 10^{22} = 0{,}325 \times 10^{23} = 0{,}0325 \times 10^{24}.$$

Znormalizowana liczba zmiennoprzecinkowa to taka, w której mantysa spełnia nierówność:

$$\mathbf{p > | m | \geq 1}$$

Czyli postać znormalizowana wyżej wypisanej liczby będzie $3{,}25 \times 10^{22}$ (p=10 ,m = 3,25 )

*Przykładowy program przeliczający wartość liczby zmiennoprzecinkowej **dodatniej** z dowolnego systemu (od dwójkowego do szestnastkowego) na system dziesiętny.*

```
Program Liczby_zmiennoprzecinkowe;
{ program przelicza liczby zmiennoprzecinkowe z dowolnego systemu }
{ (od 2 do 16) na system dziesietny.                              }
{ przy wprowadzaniu liczby do przeliczenia nalezy podac :         }
{ podstawe (zmienna p), ceche (zmienna c), mantyse (zmienna m)    }
{ Uwaga : program NIE sprawdza poprawnosci wprowadzonej liczby    }
{         i dziala tylko dla liczb dodatnich                      }

Uses Crt;

Var c,m,p,calk,ulamek : String;
   s,p_,w           : LongInt;
   c_               : Integer;
   mantysa_calk,
   mantysa_ulamk    : Real;
   liczba           : Extended;
   {-----------------------------------------------------------------}
      { Procedura znajdujaca czesc calkowita i ulamkowa liczby }
      Procedure Znajdz_Ulamek (liczba : String;
                      Var calkowita,ulamkowa : String);
      Var i : Integer;
      Begin
         i := 1;
         While i <= Length (liczba) do
         Begin
            If (liczba[i] =',') or (liczba[i] ='.') Then
            Begin
              calkowita := Copy (liczba,1,i-1);
              ulamkowa  := Copy (liczba,i+1,Length(liczba)-1);
              i := Length (liczba);
            End
            Else
            Begin
               calkowita := liczba;
               ulamkowa  := '';
            End;
```

```
            Inc(i);
          End;
        End;
    {-----------------------------------------------------------------}
          Function Przypisz_Wartosc (znak : Char) : Integer;
          Var code,x : Integer;
          Begin
            znak := UpCase(znak);
            Case znak of
                'A' : x := 10;
                'B' : x := 11;
                'C' : x := 12;
                'D' : x := 13;
                'E' : x := 14;
                'F' : x := 15;
                Else Val (znak,x,code);
                 End;
            Przypisz_Wartosc := x;
          End;
    {-----------------------------------------------------------------}

    Function Oblicz (liczba : String; Var waga : LongInt) : LongInt;
    Var L,i,x : LongInt;
    Begin
        waga := 1*s;
        L := Przypisz_Wartosc (liczba[1]);
        For i := 2 to Length(liczba) Do
        Begin
            x := Przypisz_Wartosc (liczba[i]);
            L := L * s + x;
            waga := s*waga;
        End;
        Oblicz := L;
    End;
    {-----------------------------------------------------------------}
    { Funkcja obliczajaca potege liczby a do n (a^n)                  }
    Function Oblicz_Potege (a,n : Integer) : Real;
    Var p : Real;
    Begin
        p := 1;
        While n > 0 Do
        Begin
            p := p * a;
            n:=n-1;
        End;
        While n < 0 Do
        Begin
            p := p / a;
            n:=n+1;
        End;
        Oblicz_Potege := p;
    End;
    {-----------------------------------------------------------------}

Begin { program glowny }
    ClrScr;
    Writeln;
    Writeln ('--------------------------------------------------------------------------');
    Writeln ('    Program zamieniajacy liczbe zmiennoprzecinkowa podana w dowolnym systemie ');
    Writeln ('            ( 2 ... 16)  na wartosc w systemie dziesietnym  ');
    Writeln ('Liczbe zmiennoprzecinkowa nalezy podac w formacie : m*p^c(s), np. 2,34*10^12(3)');
    Writeln ('    gdzie m - mantysa (tu : 2,34), p - podstawa (tu : 10), c - cecha (tu : 12)');
```

```
   Writeln ('    s - podstawa systemu ( tu : 3 - system tr˘jkowy) ');
   Writeln ('---------------------------------------------------------------------------');
   Writeln;
   Write (' Podaj podstawe systemu liczby przeliczanej (od 2 do 16) s = ');
   Readln (s);
   Write (' Podaj mantyse liczby przeliczanej  m = ');
   Readln (m);
   Write (' Podaj podstawe liczby przeliczanej p = ');
   Readln (p);
   Write (' Podaj ceche liczby przeliczanej    c = ');
   Readln (c);
   calk :='';
   ulamek := '';
   Znajdz_Ulamek (m,calk,ulamek);
   { Obliczenie wartosci czesci calkowitej }
   mantysa_calk := Oblicz (calk,w);
   { Obliczenie wartosci czesci ulamkowej }
   mantysa_ulamk := Oblicz (ulamek,w);
   mantysa_ulamk := mantysa_ulamk/w;
   { obliczanie wartosci podstawy }
   p_ := Oblicz (p,w);
   { obliczenie wartosci cechy }
   c_ := Oblicz (c,w);
   If c[1] ='-' Then c_:=c_*(-1);
   { obliczenie wartosci liczby ze wzoru L=m*p^c }
   liczba := (mantysa_calk+mantysa_ulamk)* Oblicz_Potege (p_,c_);
   Writeln;
   Writeln (' Wartosc liczby ',m,'*',p,'^',c,'(',s,') wynosi ',liczba:18:0,'(10)');
   Writeln;
   Writeln (' Nacisnij dowolny klawisz');
   Repeat Until KeyPressed;
End.
```

## Algorytm przeliczania liczby dziesiętnej na liczbę zmiennoprzecinkową w innym systemie pozycyjnym

1. Obliczamy mantysę przy cesze równej 0. W tym celu wystarczy przeliczyć daną liczbę dziesiętną, na liczbę w systemie docelowym.
2. Normalizujemy mantysę modyfikując przy tym odpowiednio cechę liczby
3. Koniec

*Przykład : zapisać liczbę dziesiętną 1275,125 jako zmiennoprzecinkową liczbę w systemie czwórkowym.*

Przeliczamy liczbę 1275,125 na system czwórkowy. Robimy to osobno dla części całkowitej i ułamkowej:

| | | | |
|---|---|---|---|
| 1275 | div 4 | = 318 | i reszta 3 |
| 318 | div 4 | = 79 | i reszta 2 |
| 79 | div 4 | = 19 | i reszta 3 |
| 19 | div 4 | = 4 | i reszta 3 |
| 4 | div 4 | = 1 | i reszta 0 |
| 1 | div 4 | = 0 | i reszta 1 - koniec |

$1275_{(10)} = 103323_{(4)}$

Teraz przeliczamy na system czwórkowy część ułamkową liczby:

$0,125 \times 4 = 0,5$ – cyfra 0
$0,5 \times 4 = 2,0$ – cyfra 2 i koniec ponieważ część ułamkowa wynosi 0
$0,125_{(10)} = 0,02_{(4)}$.

Łączymy ze sobą oba wyniki i otrzymujemy postać czwórkową przeliczanej liczby dziesiętnej:
1275,125(10) = 103323,02(4)
Liczbę tę zapisujemy z cechą równą 0, czyli
$103323,02 \times 10^{0}{}_{(4)}$
Normalizujemy mantysę. W tym celu przecinek należy przesunąć o 5 pozycji w lewo, zatem cecha wzrośnie do wartości 5, co w systemie czwórkowym ma zapis $11_{(4)}$ i ostatecznie:
$1275,125_{(10)} = 1,0332302 \times 10^{11}{}_{(4)}$

Zadanie to można rozwiązać również w inny sposób. Mantysę i cechę docelowej liczby zmiennoprzecinkowej możemy wyznaczyć w systemie dziesiętnym, a następnie liczby te przeliczyć na system docelowy. Korzystamy tutaj z faktu, iż przesunięcie przecinka w systemie docelowym odpowiada pomnożeniu wartości liczby przez podstawę tego systemu (przesunięcie w prawo) lub podzieleniu jej przez podstawę (przesunięcie w lewo). Zatem:

$1275,125 = 1275,125 \times 4^{0}$
$1275,125 = 318,78125 \times 4^{1}$
$1275,125 = 79,6953125 \times 4^{2}$
$1275,125 = 19,923828125 \times 4^{3}$
$1275,125 = 4,98095703125 \times 4^{4}$
$1275,125 = 1,2452392578125 \times 4^{5}$

Teraz otrzymane liczby wystarczy zamienić na system czwórkowy i mamy gotową zmiennoprzecinkową postać znormalizowaną przeliczanej liczby.

m = 1,2452392578125

Część całkowita wynosi 1, obliczamy zatem część ułamkową mantysy:

| | | |
|---|---|---|
| $0,2452392578125 \times 4 =$ | 0,98095703125 | - cyfra 0 |
| $0,98095703125 \times 4 =$ | 3,923828125 | - cyfra 3 |
| $0,923828125 \times 4 =$ | 3,6953125 | - cyfra 3 |
| $0,6953125 \times 4 =$ | 2,78125 | - cyfra 2 |
| $0,78125 \times 4 =$ | 3,125 | - cyfra 3 |
| $0,125 \times 4 =$ | 0,5 | - cyfra 0 |
| $0,5 \times 4 =$ | 2,0 | - cyfra 2 i koniec, gdyż część ułamkowa wynosi zero |

$m = 1,10332302_{(4)}$
$p = 4_{(10)} = 10_{(4)}$
$c = 5_{(10)} = 11_{(4)}$
Zatem ostatecznie:
$1275,125_{(10)} = 1,0332302 \times 10^{11}{}_{(4)}$.

*Przykład : zapisać liczbę dziesiętną 234,13 jako zmiennoprzecinkową liczbę w systemie czwórkowym.*

Postępujemy analogicznie jak w poprzednim przykładzie – przeliczamy najpierw część całkowitą liczby, a następnie część ułamkową , na koniec wyniki łączymy

234 div 4 = 58 i reszta 2
58 div 4 = 14 i reszta 2
14 div 4 = 3 i reszta 2
3 div 4 = 0 i reszta 3 - koniec

$234_{(10)} = 3222_{(4)}$

Teraz przeliczamy na system czwórkowy część ułamkową liczby:
0,13 × 4 = 0,52 – cyfra 0
0,52 × 4 = 2,08 – cyfra 2
0,08 × 4 = 0,32 – cyfra 0
0,32 × 4 = 1,28 – cyfra 1
0,28 × 4 = 1,12 – cyfra 1
0,12 × 4 = 0,48 – cyfra 0
0,48 × 4 = 1,92 – cyfra 1
0,92 × 4 = 3,68 – cyfra 3
Stwierdzam, że chyba nigdy nie osiągniemy zerowej części ułamkowej więc kończymy przybliżeniem.
$0{,}13_{(10)} = 0{,}02011013......_{(4)}$

Stąd $234{,}13_{(10)} = 3222{,}02011013...._{(4)}$

***Algorytm wyznaczania wartości liczby dziesiętnej zmiennoprzecinkowej w dowolnym systemie***

Zamienia liczbę zmiennoprzecinkową z systemu 10 na liczbę w innym systemie ( 2 ...16)
Liczbę w systemie dziesiętnym należy podać w formacie : m*p^c np. 2,34*10^12
gdzie m - mantysa (tu : 2,34), p - podstawa (tu : 10), c - cecha (tu : 12)

*Wejście:* mantysa,cecha, podstawa – łańcuch reprezentujący liczbę w systemie dziesiętnym
sys – podstawa liczby wyjściowej
*Wyjście:* mantysa,cecha, podstawa – łańcuch reprezentujący liczbę w systemie „sys"

Funkcja Przelicz (liczba_str : String) : String;

*Zmienna pomocnicze :*
r - reszta z dzielenia całkowitoliczbowego
i - zmienna pomocnicza
cz_calk,
reszta - zmienne typu całkowitego przechowujące część całkowitą liczby oraz resztę z dzielenia
cz_ulamk,
liczba - zmienne typu rzeczywistego przechowujące część ułamkową liczby oraz całą liczbę
s,

s_odwr - zmienne łańcuchowe przechowujące odpowiednio liczbę powstałą z reszt dzielenia części całkowitej oraz odwrócony łańcuch powstały z „s” oraz ewentualnej części ułamkowej liczby

krok 1 : s:= ‘’; s_odwr := ‘’;
krok 2 : Dla i:=1 , 2 ... dlugość (liczba_str) wykonaj krok 3
krok 3 : Jeśli liczba_str [ i ] = ‘,’ to liczba_str [ i ] := ‘.’; { zamiana przecinka na kropkę }
krok 4 : Zamień zmienną typu łańcuchowego (liczba_str) na zmienną typu rzeczywistego (zmienna „liczba”)
krok 5 : Podstaw pod zmienną cz_calk wartość całkowitą zmiennej „liczba”
Podstaw pod zmienną cz_ulamk wartość ułamkową zmiennej „liczba”
krok 6 : Dopóki cz_calk > 0 wykonuj :
reszta := cz_calk mod sys;
cz_calk := cz_calk div sys;
s := s+ Przypisz_Znak (reszta);
{ podstawienie pod „s_odwr” znaków z reszt dzielenia w odwrotnej kolejności }
krok 7 : Dla i := długość (s), długość (s) – 1 ..... 1 wykonuj s_odwr := s_odwr + s[ i ]
krok 8 : Jeśli cz_ulamk > 0 wykonaj kroki 9 i 10
krok 9 : i := 1; s_odwr := s_odwr + ‘ , ‘;
krok 10 : Dopóki (cz_ulamk > 0) and ( i < 11 ) powtarzaj krok 11, 12 i 13
krok 11 : liczba := cz_ulamk * sys;
krok 12 : Podstaw pod zmienną cz_calk wartość całkowitą zmiennej „liczba”
Podstaw pod zmienną cz_ulamk wartość ułamkową zmiennej „liczba”
krok 13 : i := i + 1;
s_odwr := s_odwr + Przypisz_Znak ( cz_calk );
krok 14 : Przelicz := s_odwr; { koniec funkcji Przelicz }

{ Program główny }

krok 1 : podstawa := ‘ 10 ‘;
krok 2 : Czytaj mantyse ; Czytaj ceche ; Czytaj sys;
krok 3 : mantysa := Przelicz (mantysa);
podstawa := Przelicz (podstawa);
cecha := Przelicz (cecha);
krok 4 : Wyświetl mantyse, podstawe i cechę i zakończ algorytm

Zadanie : Napisać program przeliczający wartość liczby zmiennoprzecinkowej w systemie dziesiętnym na liczbę w dowolnym systemie.

# 10. System binarny.

Historia rozwoju komputerów pokazuje nam, iż system binarny nie został od razu wybrany jako podstawowy system reprezentacji liczb w maszynach cyfrowych. Początkowo konstruktorzy próbowali stosować system dziesiętny, ponieważ był im najbliższy i bardziej zrozumiały. Jednakże system ten sprawia wiele kłopotów. Przede wszystkim operuje dziesięcioma symbolami, które należy w jakiś sposób reprezentować w maszynie cyfrowej. Prosta tabliczka dodawania czy mnożenia zawiera sto pozycji. Wszystko to znacznie komplikuje budowę komputera oraz wykonywanie obliczeń.
W latach czterdziestych XX wieku opracowywano teoretyczne podstawy działania maszyn cyfrowych i zwrócono uwagę na system binarny (dwójkowy). System ten posiada dwie cyfry - 0 i 1, które w prosty sposób można reprezentować w maszynie cyfrowej za pomocą odpowiednich napięć czy prądów elektrycznych. Układy realizujące operacje na cyfrach binarnych są nieporównywalnie prostsze od analogicznych układów operujących na cyfrach dziesiętnych. Te zalety systemu binarnego przyczyniły się do wybrania go za podstawowy system reprezentacji informacji we współczesnych komputerach.

## *10.1. BIT – podstawowa jednostka informacji.*

Słówko bit po raz pierwszy pojawiło się w literaturze informatycznej w roku 1948 w pracach znanego teoretyka informatyki Claude'a Shannona, który przyznał, iż zapożyczył ten termin od naukowca Johna Turkey'a (uważa się, iż termin software również wymyślił John). Bit powstał w trakcie drugiego śniadania, gdy John obmyślał zgrabne terminy dla pojęcia cyfry dwójkowej (binary digit): bit - binary digit, binit - binary digit, bigit - binary digit
Jak wiemy, terminy binit oraz bigit nie przyjęły się i pozostało krótkie bit. Zatem bit oznacza po prostu cyfrę binarną 0 lub 1. Cóż, trudno o coś bardziej prostego, lecz co nam to daje? W jaki sposób przy pomocy bitów można kodować informację?
Po pierwsze musimy sobie uświadomić, czym tak naprawdę jest informacja. Jest to twór abstrakcyjny, niematerialny (jeśli nie wierzysz, to odpowiedz, ile waży informacja, jaka jest w dotyku, jaki ma kolor, jak smakuje?). Przy komunikacji nie mamy bezpośrednio do czynienia z informacją, lecz z symbolami, którym tę informację przypisujemy. Dla przykładu weźmy język polski. Słowo "samolot" jest tylko ciągiem odpowiednio ze sobą połączonych dźwięków, którym nadaliśmy określone znaczenie. Również pismo to zbiór znaków, linii, które odpowiednio odczytujemy (interpretujemy przypisaną im informację).
Zatem informacja zawarta jest w symbolach - kojarzymy informację z odpowiednimi symbolami takimi jak słowa, pismo, gesty, znaki umowne (wg teorii Shannona z informacją mamy do czynienia zawsze wtedy, gdy występuje przepływ energii od nadawcy do odbiorcy - jest to definicja najbardziej ogólna). Wynika stąd wniosek, iż do przekazywania informacji potrzebujemy symboli - nośników informacji. Takim symbolem może być bit.
Bit przyjmuje dwie postacie, które (również umownie) oznaczamy odpowiednio cyfrą 0 i 1. Wyobraźmy sobie, iż cyfra 0 jest jednym symbolem, a cyfra 1 drugim (bo w rzeczywistości tak jest). Posiadamy zatem dwa symbole, którym możemy nadać dowolne, pożądane znaczenie. Jeden bit pozwoli nam przekazać informację o dwóch różnych zdarzeniach.

## *10.2. Kod binarny.*

Jeden bit to za mało, aby efektywnie kodować informację. Na szczęście możemy sobie w prosty sposób poradzić łącząc bity w grupy. Grupę taką traktujemy jak jeden symbol złożony.
Jeśli słowo binarne złożone jest z jednego bitu, to można z niego zbudować tylko dwa symbole 0 i 1. Dwa bity dają nam już cztery różne symbole : 00, 01, 10 i 11.
Dalej trzy bity pozwalają utworzyć 8 różnych symboli, a 4 bity 16 symboli.
Zauważmy, że zwiększenie długości słowa bitowego o jeden bit podwaja liczbę możliwych do utworzenia symboli. Możemy zatem napisać:

| **Liczba bitów** | **Liczba symboli** | **Potęga liczby 2** |
|---|---|---|
| 1 | 2 | $2^1$ |
| 2 | 4 | $2^2$ |
| 3 | 8 | $2^3$ |
| 4 | 16 | $2^4$ |
| 5 | 32 | $2^5$ |
| 6 | 64 | $2^6$ |
| 7 | 128 | $2^7$ |
| 8 | 256 | $2^8$ |
| ... | ... | ... |
| 16 | 65536 | $2^{16}$ |
| ... | ... | ... |
| 32 | 4294967296 | $2^{32}$ |
| ... | ... | ... |
| n | 2n | $2^n$ |

Wynika stąd prosty wniosek: dla dowolnej skończonej ilości informacji zawsze można dobrać słówka binarne o takiej ilości bitów, aby utworzyć z nich pożądaną liczbę symboli. W ten sposób powstaje kod binarny. Teraz wystarczy otrzymanym symbolom binarnym nadać znaczenia i już możemy ich używać w ten sam sposób, co słów języka polskiego.

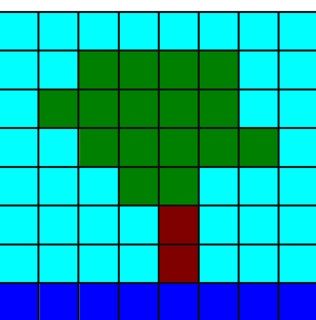
Uwaga :

- n bitów tworzy 2n różnych symboli binarnych.
- do utworzenia n symboli binarnych, gdzie n > 1, potrzebne jest co najmniej [log2(n - 1) + 1] bitów

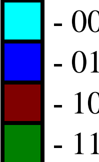
## *10.3.Zastosowanie kodów binarnych.*

### *10.3.1. Kodowanie grafiki.*

Załóżmy, iż chcemy zakodować binarnie obrazek pokazany poniżej: Jest on złożony z różnokolorowych punktów, które nazywamy pikselami (z języka ang. picture element - element obrazu, punkt).



Od razu zauważamy, że punkty są tylko w czterech kolorach. Układamy tablicę kodową kolorów, w której każdemu kolorowi punktu przyporządkujemy jeden symbol dwubitowy:




Powiązaliśmy w ten sposób informację z reprezentującymi ją symbolami. Teraz wystarczy już tylko każdy piksel zastąpić symbolem dwubitowym. W tej postaci obrazek może być przechowywany w pamięci komputera, przesyłany przez sieci teleinformatyczne oraz przetwarzany.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 0000000000000000 |
| 00 | 00 | 11 | 11 | 11 | 11 | 00 | 00 | 0000111111110000 |
| 00 | 11 | 11 | 11 | 11 | 11 | 00 | 00 | 0011111111110000 |
| 00 | 00 | 11 | 11 | 11 | 11 | 11 | 00 | 0000111111111100 |
| 00 | 00 | 00 | 11 | 11 | 00 | 00 | 00 | 0000001111000000 |
| 00 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 0000000010000000 |
| 00 | 00 | 00 | 00 | 10 | 00 | 00 | 00 | 0000000010000000 |
| 01 | 01 | 01 | 01 | 01 | 01 | 01 | 01 | 0101010101010101 |

Zwróćmy uwagę na małą czytelność dla ludzi informacji zapisanej w systemie binarnym. Szczególnie, jeśli wszystkie bity zapiszemy w jednym ciągu:
000000000000000000001111111100000011111111110000000011111111110000000011110000000000000000100000000101010101010101
Należy jednak pamiętać o tym, iż system ten jest przeznaczony dla maszyn, które nie nudzą się i nie męczą.

### *10.3.2. Kodowanie znaków.*

Zaprojektujemy kod binarny przeznaczony do kodowania małych liter alfabetu łacińskiego. W tym przypadku wiadomościami będą literki. W alfabecie łacińskim jest ich 26: abcdefghijklmnopqrstuvwxyz. Każda literka musi być kodowana innym symbolem binarnym. Musimy określić zatem niezbędną liczbę bitów tworzących te symbole. W tym celu wykorzystujemy podany wcześniej wzór i otrzymujemy:

$$[\log_2(26 - 1) + 1] = [\log_2 25 + 1] = [4,64 + 1] = [5,64] = 5 \text{ bitów}$$

5 bitów tworzy 32 symbole. Nam potrzebne jest 26, zatem 6 symboli nie zostanie wykorzystanych. Od przybytku głowa nie boli. Ważne jest, aby symboli nie było mniej niż liczba wiadomości do zakodowania. Więcej w niczym nam nie przeszkadza - nawet lepiej, gdyż w przyszłości będzie można do takiego systemu dodać 6 nowych znaków (np. spacja, przecinek, kropka itp.).

W następnym kroku układamy tabelkę kodu znakowego, w której każdej literce przydzielamy jeden symbol binarny. Po tej operacji można literki przedstawiać jako symbole binarne oraz z symboli binarnych odczytywać literki. Odwzorowanie jest zatem obustronne.

| **Binarny kod znakowy** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Znak** | **Kod** | **Znak** | **Kod** | **Znak** | **Kod** | **Znak** | **Kod** |
| a | 00000 | h | 00111 | o | 01110 | v | 10101 |
| b | 00001 | i | 01000 | p | 01111 | w | 10110 |
| c | 00010 | j | 01001 | q | 10000 | x | 10111 |
| d | 00011 | k | 01010 | r | 10001 | y | 11000 |
| e | 00100 | l | 01011 | s | 10010 | z | 11001 |
| f | 00101 | m | 01100 | t | 10011 | | |
| g | 00110 | n | 01101 | u | 10100 | | |

Wykorzystując tabelkę kodową zamieńmy na symbole binarne słowo "wagon":

w a g o n

10110 00000 00110 01110 01101

Po połączeniu bitów w jeden ciąg otrzymujemy:
1011000000001100111001101
Dla człowieka zapis ten staje się zupełnie nieczytelny, lecz komputery radzą sobie z nim znakomicie - bity to ich żywioł. Spróbujmy teraz dokonać operacji odwrotnej, tzn. odczytać słowo zakodowane w bitach, które np. otrzymaliśmy za pomocą sieci teleinformatycznej z drugiego końca świata: 1001101110010101100001110
Najpierw rozdzielamy otrzymane bity na grupy pięciobitowe: 10011 01110 01010 11000 01110

Dla każdej grupy bitów (symbolu binarnego) odszukujemy w tabelce kodu odpowiednią literkę:

10011 01110 01010 11000 01110

t o k y o

## *10.3.Bajt.*

Chociaż wolno nam tworzyć kody binarne o dowolnej liczbie bitów na symbol, to jednak z technicznego punktu widzenia wygodnie jest przyjąć pewne ustalone jednostki. Standardowe grupy bitów można w prosty sposób przechowywać w pamięciach komputerów, na nośnikach danych oraz przesyłać za pomocą sieci teleinformatycznych.

Bajt (z ang. byte) jest taką właśnie standaryzacją. Najczęściej przyjmuje się, iż jest to grupa 8 bitów. Komórki pamięci komputera przechowują informację w postaci bajtów. Również wiele urządzeń przystosowane zostało do danych przekazywanych w takiej formie (porcjami po 8 bitów) - np. drukarki, terminale, modemy, dyski elastyczne i twarde, itp. Dlatego bajt stał się kolejną po bicie jednostką informacji. Bajt utożsamiany jest ze znakiem, literą, ponieważ używa się często 8 bitowego kodu do reprezentowania znaków (ASCII - American Standard Code for Information Interchange).

**Bajt** jest grupą 8 bitów. Oznaczamy go dużą literką B w odróżnieniu od bitu - b. 1B pozwala reprezentować 256 różnych informacji.

## *10.4. Mnożniki binarne.*

W fizyce i technice stosowane są wielokrotności jednostek podstawowych. Oznaczamy je odpowiednimi przyrostkami kilo, mega, giga i tera. Podstawą tych wielokrotności jest liczba 10:

$$\begin{aligned}
\text{kilo} &= 1000 &&= 10^3 \\
\text{mega} &= 1000000 &&= 10^6 = \text{kilo} \times 1000 \\
\text{giga} &= 1000000000 &&= 10^9 = \text{mega} \times 1000 \\
\text{tera} &= 1000000000000 &&= 10^{12} = \text{giga} \times 1000
\end{aligned}$$

W systemie binarnym, ze względu na podobieństwo, zastosowano również podobne mnożniki, jednakże podstawą ich jest liczba 2, nie 10, gdyż tak jest dużo wygodniej. Starano się przy tym, aby mnożnik binarny był jak najbliższy odpowiednikowi dziesiętnemu. I tak otrzymano:

$$\begin{aligned}
\text{Kilo} &= 1024 &&= 2^{10} \\
\text{Mega} &= 1048576 &&= 2^{20} = \text{Kilo} \times 1024 \\
\text{Giga} &= 1073741824 &&= 2^{30} = \text{Mega} \times 1024 \\
\text{Tera} &= 1099511627776 &&= 2^{40} = \text{Giga} \times 1024
\end{aligned}$$

Dla odróżnienia mnożników binarnych od dziesiętnych zapisujemy je dużą literką. Jednostki binarne dzielimy na bitowe (podstawą jest bit) oraz bajtowe (podstawą jest bajt).

| Jednostki binarne | | | |
|---|---|---|---|
| bitowe | | bajtowe | |
| **b** | bit | **B** | bajt |
| **Kb** | kilobit | **KB** | kilobajt |
| **Mb** | megabit | **MB** | megabajt |
| **Gb** | gigabit | **GB** | gigabajt |
| **Tb** | terabit | **TB** | terabajt |

**Uwaga :** Podstawą mnożników binarnych jest liczba 2, a nie 10. Więc zamiast $1000 = 10^3$ stosujemy $1024 = 2^{10}$.

Zadanie : Napisać program, który dla zadanej liczby bitów (zmienna „n") wygeneruje wszystkie słowa binarne możliwe do utworzenia z zadanej ilości bitów ( np. dla n= 3 , generowane są słowa : 000, 001, 010, 100, 011, 101, 110, 111 ).

# 11. Naturalny system dwójkowy.

## *11.1. Wartość liczby w naturalnym systemie dwójkowym.*

Naturalny system dwójkowy (ang. NBS - Natural Binary System) jest najprostszym systemem pozycyjnym, w którym podstawa p = 2. System posiada dwie cyfry 0 i 1, zatem można je kodować bezpośrednio jednym bitem informacji. Wartość dziesiętna liczby zapisanej w naturalnym kodzie binarnym dana jest wzorem (patrz : systemy pozycyjne) :

$$b_{n-1}b_{n-2}...b_2b_1b_0 = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + ... + b_22^2 + b_12^1 + b_02^0$$

gdzie :

b - bit, cyfra dwójkowa 0 lub 1

n - liczba bitów w zapisie liczby

$101011_{(2)} = 2^5 + 2^3 + 2^1 + 2^0 = 32 + 8 + 2 + 1 = 43_{(10)}$

Jest to znaczne uproszczenie w stosunku do innych systemów, gdzie musimy wykonywać mnożenia cyfr przez wagi pozycji. Tutaj albo dana waga występuje w wartości liczby (cyfra 1), albo nie występuje (cyfra 0). Nie na darmo system binarny jest najprostszym systemem pozycyjnym.

## *11.2. Zakres liczby binarnej.*

Zakres n bitowej liczby w naturalnym kodzie dwójkowym wynosi $Z_{(2)} = <0, 2^n - 1>$

*Przykład : Jaką największą liczbę dziesiętną można przedstawić przy pomocy 64 bitów?*
Odp. $2^{64} - 1 = 18446744073709551616 - 1 = 18446744073709551615$

## *11.3. Schemat Hornera dla liczb binarnych.*

Schemat Hornera pozwala obliczyć wartość liczby binarnej przy minimalnej ilości operacji arytmetycznych. W systemie binarnym schemat ten jest bardzo prosty:

*Wejście:* ciąg cyfr binarnych
*Wyjście:* W - wartość liczby reprezentowanej przez ciąg cyfr binarnych
krok 1: W := pierwsza cyfra
krok 2: dopóki są kolejne cyfry wykonujemy operację W := 2 ×W + kolejna cyfra

Operację mnożenia 2 ×W możemy zastąpić dodawaniem W + W. Dodawanie komputer wykonuje o wiele szybciej od mnożenia.

*Przykład :Obliczyć schematem Hornera wartość liczby binarnej $111010111101_{(2)}$*

cyfra 1: W = 1
cyfra 1: W = (1 + 1) + 1 = 3
cyfra 1: W = (3 + 3) + 1 = 7
cyfra 0: W = (7 + 7) + 0 = 14
cyfra 1: W = (14 + 14) + 1 = 29
cyfra 0: W = (29 + 29) + 0 = 58
cyfra 1: W = (58 + 58) + 1 = 117
cyfra 1: W = (117 + 117) + 1 = 235
cyfra 1: W = (235 + 235) + 1 = 471
cyfra 1: W = (471 + 471) + 1 = 943
cyfra 0: W = (943 + 943) + 0 = 1886
cyfra 1: W = (1886 + 1886) + 1 = 3773 - koniec

## *11.4. Przeliczanie liczb dziesiętnych na binarne.*

Kolejne od końca cyfry binarne zapisu liczby w systemie dwójkowym otrzymamy jako reszty z dzielenia tej liczby przez 2.

***Algorytm wyznaczania cyfr zapisu dwójkowego liczby***

| | |
|---|---|
| Wejście: | W - wartość liczby |
| Wyjście: | ciąg cyfr binarnych reprezentujących w systemie dwójkowym wartość W |
| krok 1: | kolejna cyfra := W mod 2; W := W div 2 |
| krok 2: | Jeśli W > 0, wróć do kroku 1 |
| krok 3: | Wyprowadź otrzymane cyfry (reszty z dzielenia) w kolejności odwrotnej do ich otrzymania |
| krok 4: | Koniec |

*Przykład : przeliczyć na system dwójkowy liczbę $582642_{(10)}$.*

| | | |
|---|---|---|
| 582642 div 2 = | 291321 | i reszta 0 |
| 291321 div 2 = | 145660 | i reszta 1 |
| 145660 div 2 = | 72830 | i reszta 0 |
| 72830 div 2 = | 36415 | i reszta 0 |
| 36415 div 2 = | 18207 | i reszta 1 |
| 18207 div 2 = | 9103 | i reszta 1 |
| 9103 div 2 = | 4551 | i reszta 1 |
| 4551 div 2 = | 2275 | i reszta 1 |
| 2275 div 2 = | 1137 | i reszta 1 |
| 1137 div 2 = | 568 | i reszta 1 |
| 568 div 2 = | 284 | i reszta 0 |
| 284 div 2 = | 142 | i reszta 0 |
| 142 div 2 = | 71 | i reszta 0 |
| 71 div 2 = | 35 | i reszta 1 |
| 35 div 2 = | 17 | i reszta 1 |
| 17 div 2 = | 8 | i reszta 1 |
| 8 div 2 = | 4 | i reszta 0 |
| 4 div 2 = | 2 | i reszta 0 |
| 2 div 2 = | 1 | i reszta 0 |
| 1 div 2 = | 0 | i reszta 1 - koniec, wynik odczytujemy w kierunku z dołu do góry |

$582642_{(10)} = 10001110001111110010_{(2)}$

Zadanie : Korzystając ze schematu Hornera napisać program, który przeliczy dowolny ciąg zer i jedynek (liczbę w systemie binarnym) na liczbę w systemie dziesiętnym. Program nie może przyjmować innych znaków niż 0 i 1.

Zadanie : Korzystając ze schematu Hornera napisać program, który przeliczy dowolny ciąg cyfr od 0 do 9 (**liczbę naturalną** w systemie dziesiętnym) na liczbę w systemie dwójkowym. Program nie może przyjmować innych znaków niż 0, 1, ... 9 .

## *11.5. Dwójkowy system stałoprzecinkowy.*

### *11.5.1. Wartość dwójkowej liczby stałoprzecinkowej.*

Metodę obliczania wartości liczb stałoprzecinkowych opisaliśmy dokładnie w jednym z wcześniejszych rozdziałów. Po przecinku wagi pozycji są kolejnymi ujemnymi potęgami podstawy, zatem:
Wartość dziesiętna stałoprzecinkowej liczby binarnej wynosi wobec tego :

$$b_{n-1}...b_1b_0\,,\ b_{-1}b_{-2}...b_{-m} = b_{n-1}2^{n-1} + ... + b_1 2^1 + b_0 2^0 + b_{-1}2^{-1} + b_{-2}2^{-2} + ... + b_{-m}2^{-m}$$

gdzie:

b - bit, cyfra dwójkowa 0 lub 1
n - liczba bitów całkowitych
m - liczba bitów ułamkowych

*Przykład : Obliczyć wartość stałoprzecinkowej liczby dwójkowej* $110101{,}111011_{(2)}$.

*Sposób 1.*

Obliczamy wartość części całkowitej sumując wagi pozycji zawierających cyfrę 1:
$110101_{(2)} = 32 + 16 + 4 + 1 = 53_{(10)}$
Identycznie obliczamy wartość części ułamkowej:
$0{,}111011_{(2)} = {}^1/_2 + {}^1/_4 + {}^1/_8 + {}^1/_{32} + {}^1/_{64} = {}^{32}/_{64} + {}^{16}/_{64} + {}^8/_{64} + {}^2/_{64} + {}^1/_{64} = {}^{59}/_{64}$
Łączymy obie części w całość otrzymując wynik:
$110101{,}111011_{(2)} = 53\ {}^{59}/_{64}$

*Sposób 2.*

Wartość części całkowitej obliczamy jak powyżej. Część ułamkową traktujemy chwilowo jak liczbę całkowitą, obliczamy jej wartość i wynik mnożymy przez wagę ostatniej pozycji liczby wejściowej:
$111011_{(2)} = 32 + 16 + 8 + 2 + 1 = 59$
Waga ostatniej pozycji wynosi ${}^1/_{64}$, zatem
$0{,}111011_{(2)} = 59 \times {}^1/_{64} = {}^{59}/_{64}$

Łączymy część całkowitą z częścią ułamkową i otrzymujemy:
110101,111011(2) = 53 $^{59}/_{64}$

*Sposób 3.*

Traktujemy część całkowitą i część ułamkową jak jedną liczbę całkowitą. Za pomocą schematu Hornera wyznaczamy wartość tej liczby, a wynik mnożymy przez wagę ostatniej pozycji liczby wejściowej: 110101,111011(2) = ?
W = 1
W = (1 + 1) + 1 = 3
W = (3 + 3) + 0 = 6
W = (6 + 6) + 1 = 13
W = (13 + 13) + 0 = 26
W = (26 + 26) + 1 = 53 - część całkowita obliczona, kontynuujemy z częścią ułamkową
W = (53 + 53) + 1 = 107
W = (107 + 107) + 1 = 215
W = (215 + 215) + 1 = 431
W = (431 + 431) + 0 = 862
W = (862 + 862) + 1 = 1725
W = (1725 + 1725) + 1 = 3451 - koniec części ułamkowej
Otrzymany wynik mnożymy przez wagę ostatniej pozycji, czyli przez $^{1}/_{64}$:
110101,111011(2).= 3451 × $^{1}/_{64}$ = 53 $^{59}/_{64}$

*Zadanie : Korzystając ze schematu Hornera napisać program, który przeliczy liczbę stałoprzecinkową zapisaną w systemie dwójkowym na liczbę w systemie dziesiętnym. Program nie może przyjmować innych znaków niż 0, 1, oraz przecinka.*

```
Program Dwa_Staloprzecinkowe_Na_10;
Uses Crt;
Var  znak : Char;
     L_2  : String;
     waga : LongInt;
     liczba : Extended;
     cyfra,
     i,code : Integer;
     ulamek : Boolean;
Begin
   ClrScr;
   Writeln;
   Writeln ('-----------------------------------------------------');
   Writeln (' Program przeliczajacy liczbe w systemie dwojkowym');
   Writeln (' staloprzecinkowym na liczbe w systemie dziesietnym');
   Writeln ('-----------------------------------------------------');
   Writeln;
   Writeln (' Podaj liczbe w systemie dw˘jkowym i nacisnij ENTER');
   Writeln; Write (' ');
   znak := #0;
   L_2  := '';
   Repeat
      znak := ReadKey;
      If znak in ['0','1','2','3','4','5','6','7','8','9',','] Then
        Begin
            L_2 := L_2 + znak;
            Write (znak);
        End
        Else Write (#7)
```

```
   Until znak = #13;
   { obliczanie wartosci liczby }
   Val (L_2[1],liczba,code);
   ulamek := False;
   waga := 1;
   For i:=2 To Length (L_2) do
   Begin
     If L_2[i] = ',' Then ulamek:= True
     Else
     Begin
        Val (L_2[i],cyfra,code);
        liczba := liczba+liczba+cyfra;
        If ulamek=True Then waga := 2*waga;
     End;
   End;
   liczba := liczba/waga;
   Writeln;Writeln;
   Writeln (' Wartosc w systemie dziesietnym : ',liczba:18:4);
   Writeln (' Nacisnij dowolny klawisz');
   Repeat Until KeyPressed;
End.
```

### *11.5.2. Zakres binarnych liczb stałoprzecinkowych.*

Jaką największą liczbę można przedstawić za pomocą n bitów całkowitych i m bitów ułamkowych, gdzie n i m są liczbami naturalnymi ?
Liczbę stałoprzecinkową możemy potraktować jako złożenie dwóch liczb - całkowitej n-bitowej oraz ułamkowej m-bitowej.
Część całkowita dla n bitów przyjmuje największą wartość równą 2n - 1.
Część ułamkowa będzie największa, gdy wszystkie jej bity ustawione zostaną na 1. Dla m bitów wartość takiej liczby wyniesie (2m - 1) / 2m (zobacz sposób 3 obliczania wartości binarnej liczby stałoprzecinkowej, gdzie wartość części ułamkowej liczby dwójkowej obliczamy jako liczbę całkowitą pomnożoną przez wagę ostatniej pozycji).

Łączymy obie części i otrzymujemy: dla **n** bitów całkowitych i **m** bitów ułamkowych największą liczbą jest : $2^n - 1 + \frac{2^m - 1}{2^m}$.

## *11.6. Operacje arytmetyczne na systemie dwójkowym.*

Zasady arytmetyki w systemie binarnym są prawie identyczne jak w dobrze nam znanym systemie dziesiętnym. Zaletą arytmetyki binarnej jest jej prostota, dzięki czemu można ją tanio realizować za pomocą układów elektronicznych.

### *11.6.1. Dodawanie.*

Do wykonywania dodawania niezbędna jest znajomość tabliczki dodawania, czyli wyników sumowania każdej cyfry z każdą inną. W systemie binarnym mamy tylko dwie cyfry 0 i 1, zatem tabliczka dodawania jest niezwykle prosta i składa się tylko z 4 pozycji:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10

Przykład : Zsumować liczby binarne $1111001_{(2)}$ oraz $10010_{(2)}$.

Sumowane liczby zapisujemy jedna pod drugą tak, aby w kolejnych kolumnach znalazły się cyfry stojące na pozycjach o tych samych wagach (identycznie postępujemy w systemie dziesiętnym zapisując liczby w słupkach przed sumowaniem):

```
  1111001
+   10010
```

Sumowanie rozpoczynamy od ostatniej kolumny. Sumujemy cyfry w kolumnie zgodnie z podaną tabelką zapisując wynik pod kreską:

```
  1111001
+   10010
     1011
```

Jeśli wynik sumowania jest dwucyfrowy (1 + 1 = 10), to pod kreską zapisujemy tylko ostatnią cyfrę 0, a 1 przechodzi do następnej kolumny - dodamy ją do wyniku sumowania cyfr w następnej kolumnie. Jest to tzw. przeniesienie (ang. carry). Przeniesienie zaznaczyliśmy na czerwono:

```
     1
  1111001
+   10010
    01011
```

Jeśli w krótszej liczbie zabrakło cyfr, to dopisujemy zera. Pamiętajmy o przeniesieniach.

```
  1 11
  1111001
+ 0010010
  0001011
```

Dodaliśmy wszystkie cyfry, ale przeniesienie wciąż wynosi 1. Zatem dopisujemy je do otrzymanego wyniku (możemy potraktować pustą kolumnę tak, jakby zawierała cyfry 0 i do wyniku sumowania dodać przeniesienie).

```
   111
  01111001
+ 00010010
  10001011
```

***UWAGA !!!:***

W pamięci komputera liczby binarne przechowywane są w postaci ustalonej ilości bitów (np. 8, 16, 32 bity). Jeśli wynik sumowania np. dwóch liczb 8 bitowych jest większy niż 8 bitów, to najstarszy bit (dziewiąty bit) zostanie utracony. Sytuacja taka nazywa się nadmiarem (ang. overflow) i występuje zawsze, gdy wynik operacji arytmetycznej jest większy niż górny zakres danego formatu liczb binarnych (np. dla 8 bitów wynik większy od $2^8$ - 1, czyli większy od 255):

$11111111_{(2)} + 00000001_{(2)} = 1|00000000_{(2)}$ (255 + 1 = 0)

Przy nadmiarze otrzymany wynik jest zawsze błędny, dlatego należy bardzo dokładnie analizować problem obliczeniowy i ustalić typy danych zgodnie z przewidywanym zakresem wartości otrzymywanych wyników. Zwykle kompilatory języków programowania posiadają opcję włączenia sprawdzania zakresów wyników operacji arytmetycznych na liczbach całkowitych (w Borland Pascalu jest to opcja menu Options - > Compiler, a w okienku opcji zaznaczamy Overflow checking). Opcję tę włączamy w fazie testowania programu. Natomiast w gotowym produkcie należy ją wyłączyć, ponieważ wydłuża czas wykonywania operacji arytmetycznych.

### *11.6.2. Odejmowanie.*

Przy odejmowaniu korzystamy z tabliczki odejmowania, która w systemie binarnym jest bardzo prosta:

0 - 0 = 0
0 - 1 = 1 i pożyczka do następnej pozycji
1 - 0 = 1
1 - 1 = 0

Odejmując 0 - 1 otrzymujemy wynik 1 i pożyczkę (ang. borrow) do następnej pozycji. Pożyczka oznacza konieczność odjęcia 1 od wyniku odejmowania cyfr w następnej kolumnie. Identycznie postępujemy w systemie dziesiętnym, tyle że tam jest to o wiele bardziej skomplikowane.
W naturalnym kodzie binarnym nie ma liczb ujemnych więc od liczb większych odejmujemy mniejsze (później zostanie pokazane, że można otrzymywać liczby ujemne).

*Przykład : Wykonać odejmowanie w systemie binarnym $1101110_{(2)}$ - $1111_{(2)}$.*

Obie liczby umieszczamy jedna pod drugą tak, aby ich cyfry znalazły się w kolumnach o tych samych wagach:

```
  1101110
-    1111
```

Odejmowanie rozpoczynamy od cyfr ostatniej kolumny. Wyniki zapisujemy pod kreską. W tym przykładzie odjęcie ostatnich cyfr 0 - 1 daje wynik 1 oraz pożyczkę do następnej kolumny. Pożyczki zaznaczamy kolorem czerwonym.

```
            1
  1 1 0 1 1 1 0
-       1 1 1 1
---------------
              1
```

Odjęcie cyfr w drugiej od końca kolumnie daje wynik 1 - 1 = 0. Od tego wyniku musimy odjąć pożyczkę 0 - 1 = 1 i pożyczka do następnej kolumny.

```
          1 1
  1 1 0 1 1 1 0
-       1 1 1 1
---------------
            1 1
```

Według tych zasad kontynuujemy odejmowanie cyfr w pozostałych kolumnach. Pamiętajmy o pożyczkach! Jeśli w krótszej liczbie zabraknie cyfr, to możemy kolumny wypełnić zerami:

```
    1 1 1 1 1
  1 1 0 1 1 1 0
- 0 0 0 1 1 1 1
---------------
  1 0 1 1 1 1 1
```

$1101110_{(2)} - 1111_{(2)} = 1011111_{(2)}$ $(110_{(10)} - 15_{(10)} = 95_{(10)})$.

*UWAGA :* Przy odejmowaniu również może dochodzić do nieprawidłowych sytuacji. Jeśli od liczby mniejszej odejmiemy mniejszą, to wynik będzie ujemny. Jednakże w naturalnym systemie binarnym nie można zapisywać liczb ujemnych. Zobaczmy zatem co się stanie, gdy od liczby 0 odejmiemy 1, a wynik ograniczymy do 8 bitów:

```
 1 1 1 1 1 1 1 1
   0 0 0 0 0 0 0 0
-  0 0 0 0 0 0 0 1
-------------------
   1 1 1 1 1 1 1 1
```

Otrzymujemy same jedynki, a pożyczka nigdy nie zanika. Sytuacja taka nazywa się niedomiarem (ang. underflow) i występuje zawsze, gdy wynik operacji arytmetycznej jest mniejszy od dolnego zakresu formatu liczb binarnych (dla naturalnego kodu dwójkowego wynik mniejszy od zera). Oczywiście otrzymany rezultat jest błędny.

### *11.6.3. Mnożenie*

Mnożenie binarne należy rozpocząć od tabliczki mnożenia.

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Tabliczka mnożenia binarnego (podobnie jak w systemie dziesiętnym) posłuży do tworzenia iloczynów częściowych cyfr mnożnej przez cyfry mnożnika. Iloczyny te następnie dodajemy według zasad opisanych poniżej i otrzymujemy wynik mnożenia.

*Przykład : Pomnożyć binarnie liczbę $1101_{(2)}$ przez $1011_{(2)}$.*

Obie liczby umieszczamy jedna pod drugą tak, aby ich cyfry znalazły się w kolumnach o tych samych wagach:

```
     1101
x    1011
```

Każdą cyfrę mnożnej mnożymy przez poszczególne cyfry mnożnika zapisując wyniki mnożeń w odpowiednich kolumnach - tak samo postępujemy w systemie dziesiętnym, a tutaj jest nawet prościej, gdyż wynik mnożenia cyfry przez cyfrę jest zawsze jednocyfrowy:

```
          1 1 0 1
x         1 0 1 1
-----------------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
```

Zwróć uwagę, iż wynikiem mnożenia mnożnej przez cyfrę mnożnika jest powtórzenie mnożnej z przesunięciem o pozycję cyfry (cyfra mnożnika 1) lub same zera (cyfra mnożnika 0). Spostrzeżenie to bardzo ułatwia konstrukcję układów mnożących.

Puste kolumny uzupełniamy zerami i dodajemy do siebie wszystkie cyfry w kolumnach (wiersz z samymi zerami – kolor niebieski, zosłał pominięty) . Uwaga na przeniesienia !

```
        1 1 0 1
x       1 0 1 1
---------------
  0 0 0 1 1 0 1
  0 0 1 1 0 1 0
+ 1 1 0 1 0 0 0
---------------
1 0 0 0 1 1 1 1
```

***UWAGA :*** Z uwagi na ustalone formaty danych binarnych w komputerach (8, 16 i 32 bity) mnożenie również może dawać niepoprawne rezultaty, gdy wynik będzie większy od górnego zakresu liczb dla danego formatu, czyli od max = $2^n - 1$, gdzie n - liczba bitów w danym formacie.

### *11.6.4. Dzielenie.*

Dzielenie binarne jest najbardziej skomplikowaną operacją arytmetyczną z dotychczas opisywanych. Wymyślono wiele algorytmów efektywnego dzielenia, ale dla potrzeb tego opracowania wystarczy znany wam algorytm szkolny, który polega na cyklicznym odejmowaniu odpowiednio przesuniętego dzielnika od dzielnej. W systemie dwójkowym jest to szczególnie proste, ponieważ dzielnika nie musimy mnożyć.

*Przykład : Podzielimy liczbę* $1101_{(2)}$ *przez* $10_{(2)}$ $(13_{(10)} : 2_{(10)})$.

Przesuwamy w lewo dzielnik, aż zrówna się jego najstarszy, niezerowy bit z najstarszym, niezerowym bitem dzielnej. Nad dzielną rysujemy kreseczkę (kolor czerwony):

```
____
1101    - dzielna
10      - przesunięty dzielnik
```

Porównujemy dzielną z dzielnikiem. Jeśli dzielna jest większa lub równa dzielnikowi, to odejmujemy od niej dzielnik. Ponad kreską na pozycji ostatniej cyfry dzielnika piszemy 1. Jeśli dzielna jest mniejsza od dzielnika, to nie wykonujemy odejmowania, lecz przesuwamy dzielnik o 1 pozycję w prawo i powtarzamy opisane operacje. Jeśli w ogóle dzielnika nie da się odjąć od dzielnej (np. przy dzieleniu 7 przez 9), to wynik dzielenia wynosi 0, a dzielna ma w takim przypadku wartość reszty z dzielenia. W naszym przykładzie odejmowanie to jest możliwe, zatem:

```
   1       - pierwsza cyfra wyniku dzielenia
 -------
 1 1 0 1   - dzielna
- 1 0      - przesunięty dzielnik
 -------
 0 1 0 1   - wynik odejmowania dzielnika od dzielnej
```

Dzielnik przesuwamy o jeden bit w prawo i próbujemy tego samego z otrzymaną różnicą. Jeśli odejmowanie jest możliwe, to nad kreską w następnej kolumnie dopisujemy 1, odejmujemy dzielnik od różnicy, przesuwamy go o 1 bit w prawo i kontynuujemy. Jeśli odejmowanie nie jest możliwe, to dopisujemy nad kreską 0, przesuwamy dzielnik o 1 bit w prawo i kontynuujemy.

110 - wynik dzielenia
1101 - dzielna
-10 - przesunięty dzielnik
0101 - dzielna po pierwszym odejmowaniu przesuniętego dzielnika
- 10 - przesunięty dzielnik
0001 - dzielna po drugim odejmowaniu przesuniętego dzielnika
- 10 - dzielnik na swoim miejscu, odejmowanie niemożliwe
**0001** - reszta z dzielenia

Operacje te wykonujemy dotąd, aż dzielnik osiągnie swoją pierwotną wartość. Pozostała dzielna jest resztą z dzielenia. Oczywiście w tym momencie możemy dalej kontynuować odejmowanie wg opisanych zasad otrzymując kolejne cyfry ułamkowe - identycznie postępujemy w systemie dziesiętnym. Jednakże pozostawimy ten problem do rozwiązania bardziej ambitnym czytelnikom.

W naszym przykładzie otrzymaliśmy wynik dzielenia równy:
$1101_{(2)} : 10_{(2)} = 110_{(2)}$ i resztę $1_{(2)}$ $(6_{(10)}$ i $1_{(10)})$
Jest to wynik poprawny, gdyż 2 mieści się w 13 sześć razy i pozostaje reszta 1.

## *11.7. Szybkie potęgowanie liczb.*

Binarna reprezentacja liczb naturalnych jest podstawą dla szybkich metod obliczania wartości potęgi $x^m$, gdzie n jest liczbą naturalną, zaś x dowolną liczbą rzeczywistą. Metody te znajdują zastosowanie w algorytmach kryptograficznych, w których są obliczane wartości potęg o bardzo dużych wykładnikach.

Posłużymy się najpierw przykładem. Przypuśćmy, że chcemy obliczyć wartość potęgi $x^{22}$.
Proste wymnożenie podstawy przez siebie $x^{22} = x \bullet x \bullet ... \bullet x$ to 21 mnożeń. Ale skorzystajmy z binarnej reprezentacji wykładnika potęgi. Można go przedstawić jako sumę potęg liczby 2:

$$22 = 2 + 4 + 16 = 2^1 + 2^2 + 2^4$$

wtedy nasza potęga przyjmuje postać :

$$x^{22} = x^{2+4+16} = x^2 \bullet x^4 \bullet x^{16}$$

i można ją obliczyć wielokrotnie, podnosząc do kwadratu podstawę x i mnożąc przez siebie odpowiednie czynniki. W naszym przypadku obliczamy:
$x^4 = (x^2)^2$, $x^8 = (x^4)^2$, $x^{16} = (x^8)^2$ i mnożymy przez siebie $x^2x^4x^{16}$. W sumie wykonujemy 6 mnożeń.

Sposób 2.
Korzystając z przedstawienia wykładnika w reprezentacji binarnej $22 = (10110)_2$ w postaci schematu Hornera, wykładnik można zapisać :
$22 = 1\bullet 2^4 + 0\bullet 2^3 + 1\bullet 2^2 + 1\bullet 2^1 + 0\bullet 2^0 = (((2 + 0)\bullet 2 +1)\bullet 2 + 1)\bullet 2 + 0$, a stąd otrzymujemy:

$$x^{(((2+0)\bullet 2+1)\bullet 2+1)\bullet 2+0} = x^{(((2+0)\bullet 2+1)\bullet 2+1)\bullet 2} = (x^{((2+0)\bullet 2+1)\bullet 2+1})^2 = (x^{((2+0)\bullet 2+1)\bullet 2}\, x)^2 =$$
$$= (x^{((2+0)\bullet 2+1)})^2\, x)^2 = (x^{((2+0)\bullet 2}\, x)^2\, x)^2 = (x^{((2+0)})^2\, x)^2\, x)^2 = (x^2)^2\, x)^2\, x)^2$$

Korzystając z tego zapisu, potęgę $x^{22}$ obliczamy wykonując kolejno następujące mnożenia :
$x^2$, $x^4 = (x^2)^2$, $x^5 = x^4 x$, $x^{10} = (x^5)^2$, $x^{11} = x^{10} x$, $x^{22} = (x^{11})^2$ . W sumie również też 6 mnożeń.

Obie metody korzystają z binarnego rozwinięcia wykładnika, ale różnią się tym, że w pierwszym przypadku to rozwinięcie jest przeglądane od najmniej znaczącego bitu (mówimy o metodzie od prawej do lewej), a w drugim -od najbardziej znaczącego (czyli metodą od lewej do

prawej). Ponieważ reprezentacja binarna liczby naturalnej jest tworzona od najmniej znaczącego bitu, podamy teraz algorytm obliczania wartości potęgi, który wykonuje potęgowanie, rozkładając wykładnik na postać binarną.

*Specyfikacja problemu algorytmicznego.*

*Dane wejściowe :*
m - wykładnik potęgi $m \in N$ ,
x - dowolna liczba podnoszona do potęgi

*Dane wyjściowe :*
y - wartość potęgi $x^m$

*Zmienne pomocnicze :*
z,L - zmienne pomocnicze

***Lista kroków***

Krok 1 : y:= 1; L := m ; z := x;
Krok 2 : Jeśli L mod 2 = 1 { czy jest to liczba nieparzysta }to y := y * z;
Krok 3 : L:= L div 2;
Krok 4 : Jeśli L <> 0 to z := z * z i przejdź do kroku 2
Krok 5 : Wypisz y i zakończ algorytm

## 12. Kodowanie liczb binarnych ze znakiem.

W poprzednich rozdziałach został omówiony naturalny system binarny NBC (ang. Natural Binary Code). Jest on niezmiernie ważny dla informatyka, ponieważ stanowi punkt wyjścia większości innych systemów zapisu liczb. Jednakże NBC pozwala zapisywać tylko liczby dodatnie oraz liczbę zero. Na pierwszy rzut oka można zapisać w nim także liczby ujemne – dopisać przed liczbą znak minus, na przykład: $-101_{(2)} = -5_{(10)}$.

Ale tak można zrobić jedynie na papierze. Liczby binarne muszą być umieszczane w pamięci komputera, a tam mieszczą się tylko bity, czyli symbole 0 lub 1. Żaden bit nie przyjmuje stanu '-'. Mamy więc problem. A problem ten sprowadza się do wymyślenia takiego sposobu kodowania, aby za pomocą bitów można było zapisywać wartości ujemne. Sposobów wymyślono kilka i opiszemy je dokładnie w kolejnych podrozdziałach.

O ile system NBC był swobodny co do ilości bitów w zapisie liczby, to systemy kodowania liczb ze znakiem posiadają ściśle ustalone formaty, tzn. ilość bitów w zapisie liczby jest określona na stałe (8b, 16b, 32b, 64b itd.). Jest to konieczne, ponieważ zwykle najstarszy bit posiada inne znaczenie od pozostałych bitów liczby (nazywany jest on często bitem znaku - ang. sign bit).

## *12.1. Zapis „znak-moduł”.*

Koncepcyjnie zapis znak - moduł (w skrócie ZM - ang. SM Signed Magnitude) jest najbardziej zbliżony do systemu zapisu liczb używanego przez nas samych. Liczba ZM składa się z dwóch części - bitu znaku oraz bitów wartości liczby (modułu):

$$b_{n-1}\ b_{n-2}\ b_{n-3} \ldots b_2\ b_1\ b_0$$

$b_{n-1}$ - bit znaku liczby
$b_{n-2} \ldots b_0$ - bity modułu liczby

Dla liczb dodatnich i zera bit znaku ma wartość 0, dla liczb ujemnych i zera ma wartość 1. Format zapisu ZM musi być ściśle ustalony, aby wiadomo było, który bit jest bitem znaku - w operacjach arytmetycznych bit znaku musimy traktować inaczej niż inne bity.

### *12.1.1. Wartość dziesiętna liczby w zapisie ZM.*

Moduł liczby ZM jest zapisany w naturalnym kodzie dwójkowym NBC, zatem w celu obliczenia jej wartości moduł mnożymy przez 1, gdy bit znaku wynosi 0 lub przez -1, gdy bit znaku wynosi 1. Wzór jest następujący: $L_{ZM} = (-1)^{\text{bit znaku}}$ x moduł liczby

Rozpisując poszczególne bity otrzymujemy:

$$b_{n-1}b_{n-2}\ldots b_2b_1b_0 = (-1)^{b_{n-1}} \times (b_{n-2}2^{n-2} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0)$$

gdzie
b - bit, cyfra dwójkowa 0 lub 1
n - liczba bitów w zapisie liczby

| 4 bitowy system ZM | | |
|---|---|---|
| **Kod ZM** | **Przeliczenie** | **Wartość** |
| 0000 | $(-1)^0 \times 0$ | 0 |
| 0001 | $(-1)^0 \times (2^0)$ | 1 |
| 0010 | $(-1)^0 \times (2^1)$ | 2 |
| 0011 | $(-1)^0 \times (2^1 + 2^0)$ | 3 |
| 0100 | $(-1)^0 \times (2^2)$ | 4 |
| 0101 | $(-1)^0 \times (2^2 + 2^0)$ | 5 |
| 0110 | $(-1)^0 \times (2^2 + 2^1)$ | 6 |
| 0111 | $(-1)^0 \times (2^2 + 2^1 + 2^0)$ | 7 |
| 1000 | $(-1)^1 \times 0$ | 0 |
| 1001 | $(-1)^1 \times (2^0)$ | (-1) |
| 1010 | $(-1)^1 \times (2^1)$ | (-2) |
| 1011 | $(-1)^1 \times (2^1 + 2^0)$ | (-3) |
| 1100 | $(-1)^1 \times (2^2)$ | (-4) |
| 1101 | $(-1)^1 \times (2^2 + 2^0)$ | (-5) |
| 1110 | $(-1)^1 \times (2^2 + 2^1)$ | (-6) |
| 1111 | $(-1)^1 \times (2^2 + 2^1 + 2^0)$ | (-7) |

W tabelce obok przedstawiono wszystkie możliwe do utworzenia liczby w 4-ro bitowym kodzie ZM. Zwróć uwagę, iż wartość 0 posada dwa słowa kodowe: 0000 oraz 1000. W obu przypadkach wartość modułu wynosi 0.
Liczby ujemne uzyskujemy przez ustawienie bitu znaku na 1.
Zatem :

$3_{(10)} = 0011_{(ZM)}$ ,
$-3_{(10)} = 1011_{(ZM)}$.

*Przykład : Obliczyć wartość dziesiętną liczby $10110111_{(ZM)}$.*

Pierwszy bit zapisu ZM jest bitem znaku. Wartość 1 informuje nas, iż jest to liczba ujemna. Pozostałe bity tworzą wartość liczby. Moduł jest zapisany w naturalnym systemie dwójkowym, zatem:

$$10110111_{(ZM)} = (-1)^1 \times (25 + 24 + 22 + 21 + 20)$$
$$10110111_{(ZM)} = - (32 + 16 + 4 + 2 + 1)$$
$$10110111_{(ZM)} = - 55_{(10)}$$

Kod „znak-moduł" nie ma zbyt wielkiego zastosowania w praktyce i dlatego nie będziemy się nim dalej zajmować. Został on omówiony tylko ze względu na jego podobieństwo do zapisu liczb w systemie dziesiętnym. Warto tu jednak wspomnieć, że w „ZM" istnieje możliwość zapisu liczb stałoprzecinkowych. Wadą kodu ZM jest duża modyfikacja reguł arytmetyki NBC.

*Przykład : Wyznacz zapis ZM liczby dziesiętnej -3,75. Format ZM jest 8-mio bitowy. Moduł posiada cztery cyfry ułamkowe.*

1. Liczba -3,75 jest ujemna, zatem bit znaku wynosi 1.
2. Moduł liczby jest równy 3,75.
3. Wyznaczamy dwójkowy zapis modułu: $3{,}75_{(10)} = 11{,}11_{(2)}$ (patrz : rozdział 11.5.1)
4. W podanym formacie ZM moduł jest 7-mio bitowy z 4-ema bitami ułamkowymi. Otrzymany zapis dwójkowy musimy zatem odpowiednio uzupełnić bitami o wartości 0, aby był zgodny z formatem: 11,11 = 011,1100
5. Do tak otrzymanego modułu dodajemy bit znaku otrzymując: $-3{,}75_{(10)} = 1011{,}1100_{(ZM)}$.

## *12.2. Zapis uzupełnień do 1 – U1 (1C - One's Complement)*

Drugim podejściem do rozwiązania problemu liczb ze znakiem jest system uzupełnień do 1 zwany systemem U1 (ang. 1C - One's Complement). W systemie tym wszystkie bity zapisu liczby posiadają swoje wagi (w ZM bit znaku nie posiadał wagi). Najstarszy bit jest bitem znaku i ma wagę równą (-2n-1+1), gdzie n oznacza ilość bitów w zapisie liczby. Pozostałe bity posiadają wagi takie jak w naturalnym systemie dwójkowym (patrz : rozdział 11).

| Wartości wag pozycji w zapisie U1 | | | | | | | |
|---|---|---|---|---|---|---|---|
| **waga** | $-2^{n-1}+1$ | $2^{n-2}$ | $2^{n-3}$ | | $2^2$ | $2^1$ | $2^0$ |
| **cyfra** | $b_{n-1}$ | $b_{n-2}$ | $b_{n-3}$ | **...** | $b_2$ | $b_1$ | $b_0$ |

Bit znaku przyjmuje wartość 0 dla liczb dodatnich, a dla liczb ujemnych wartość 1 (ponieważ tylko waga tej pozycji jest ujemna, to jest to jedyny sposób na otrzymanie wartości ujemnej).

Wartość liczby U1 obliczamy zgodnie z podanymi wcześniej zasadami - cyfry mnożymy przez wagi pozycji, na których się znajdują i dodajemy otrzymane iloczyny:

$$b_{n-1}b_{n-2}b_{n-3}...b_2b_1b_{0\,(U1)} = b_{n-1}(-2^{n-1}+1) + b_{n-2}2^{n-2} + b_{n-3}2^{n-3} + ... + b_22^2 + b_12^1 + b_02^0$$

gdzie

b - bit, cyfra dwójkowa 0 lub 1
n - liczba bitów w zapisie liczby

| 4 bitowy system U1 | | |
|---|---|---|
| **Kod U1** | **Przeliczenie** | **Wartość** |
| 0000 | 0 | 0 |
| 0001 | $2^0$ | 1 |
| 0010 | $2^1$ | 2 |
| 0011 | $2^1 + 2^0$ | 3 |
| 0100 | $2^2$ | 4 |
| 0101 | $2^2 + 2^0$ | 5 |
| 0110 | $2^2 + 2^1$ | 6 |
| 0111 | $2^2 + 2^1 + 2^0$ | 7 |
| 1000 | $(-2^3 + 1)$ | (-7) |
| 1001 | $(-2^3 + 1) + 2^0$ | (-6) |
| 1010 | $(-2^3 + 1) + 2^1$ | (-5) |
| 1011 | $(-2^3 + 1) + 2^1 + 2^0$ | (-4) |
| 1100 | $(-2^3 + 1) + 2^2$ | (-3) |
| 1101 | $(-2^3 + 1) + 2^2 + 2^0$ | (-2) |
| 1110 | $(-2^3 + 1) + 2^2 + 2^1$ | (-1) |
| 1111 | $(-2^3 + 1) + 2^2 + 2^1 + 2^0$ | 0 |

Ponieważ w systemach binarnych cyfry przyjmują tylko dwie wartości 0 lub 1, można rachunek znacznie uprościć sumując jedynie te wagi, dla których cyfra zapisu liczby ma wartość 1.

W tabelce obok przedstawiliśmy wszystkie możliwe do utworzenia liczby w 4-ro bitowym kodzie U1. Zwróć uwagę, iż wartość 0 posada dwa słowa kodowe: 0000 oraz 1111. Wynika to bezpośrednio ze wzoru obliczania wartości liczby U1. Jeśli wszystkie cyfry mają wartość 0, to żadna waga pozycji nie uczestniczy w wartości liczby i wartość ta jest równa 0.
Z kolei jeśli bit znaku jest równy 1, to jego waga wlicza się do wartości liczby. Waga bitu znakowego jest równa $(-2^{n-1}+1)$. Dla 4 bitowego kodu n=4, zatem waga bitu znakowego wynosi:
waga $b_3 = (-2^3+1) = -7$
Jeśli pozostałe bity liczby są ustawione na 1, to ich wagi sumują się do wartości 7. Jeśli teraz dodamy wagę bitu znakowego i wagi pozostałych bitów, otrzymamy wartość 0.

Drugą charakterystyczną cechą kodu U1 są liczby przeciwne. Zwróć uwagę, iż liczba przeciwna zawsze powstaje w kodzie U1 przez negację wszystkich bitów:

$1_{(10)} = 0001_{(U1)}$ $\quad$ $(-1)_{(10)} = 1110_{(U1)}$
$5_{(10)} = 0101_{(U1)}$ $\quad$ $(-5)_{(10)} = 1010_{(U1)}$
$7_{(10)} = 0111_{(U1)}$ $\quad$ $(-7)_{(10)} = 1000_{(U1)}$

Zasada ta obowiązuje dla kodów U1 o dowolnej długości.

### *12.2.1. Przeliczanie liczb dziesiętnych na zapis U1.*

Jeśli liczba jest dodatnia, to najstarszy bit znakowy posiada wartość 0. Pozostałe bity służą do zapisu liczby w naturalnym kodzie binarnym:
$0111_{(U1)} = 7_{(10)}$, $\quad$ $0001_{(U1)} = 1_{(10)}$, $\quad$ $01111111_{(U1)} = 127_{(10)}$

Jeśli liczba jest ujemna, to najstarszy bit znakowy ma wartość 1. Pozostałe bity są negacjami bitów modułu wartości liczby:
$1101_{(U1)} = (-2)_{(10)} \rightarrow$ moduł $2_{(10)} = 010_{(2)} \rightarrow$ NOT 010 = 101
$1100_{(U1)} = (-3)_{(10)} \rightarrow$ moduł $3_{(10)} = 011_{(2)} \rightarrow$ NOT 011 = 100
$1010_{(U1)} = (-5)_{(10)} \rightarrow$ moduł $5_{(10)} = 101_{(2)} \rightarrow$ NOT 101 = 010

Wynika stąd prosta metoda przeliczania liczby dziesiętnej na zapis U1.

***Sposób 1 wyznaczania zapisu U1 dla liczby dziesiętnej***

1) Jeśli liczba jest dodatnia, znajdujemy jej reprezentację w naturalnym kodzie binarnym i uzupełniamy bitami o wartości 0 do uzyskania zadanej liczby bitów wymaganej przez przyjęty format zapisu U1.
2) Jeśli liczba jest ujemna, obliczamy jej moduł. Moduł przedstawiamy w naturalnym systemie dwójkowym uzupełniając go bitami o wartości 0 do długości przyjętego formatu U1. Następnie wszystkie bity zamieniamy na przeciwne i w wyniku otrzymujemy zapis U1.

*Przykład : Wyznaczyć 8-mio bitowy zapis U1 liczby dziesiętnej 76.*

Liczba 76 jest dodatnia, zatem wyznaczamy jej zapis w naturalnym systemie dwójkowym:

$$76_{(10)} = 1001100_{(2)}$$

Otrzymaną liczbę dwójkową uzupełniamy bitami o wartości 0 do długości formatu otrzymując:

$$76_{(10)} = 01001100_{(U1)}.$$

*Przykład : Wyznaczyć 8-mio bitowy zapis U1 liczby dziesiętnej (-113).*

Liczba (-113) jest ujemna. Jej moduł wynosi 113. Wyznaczamy zapis dwójkowy modułu:

$$113_{(10)} = 1110001_{(2)}$$

Otrzymany zapis uzupełniamy bitami 0 do długości 8 bitów. Następnie negujemy wszystkie bity i otrzymujemy w ten sposób zapis U1 liczby -113:

$$-113_{(10)} = \text{NOT } 01110001 = 10001110_{(U1)}.$$

***Sposób 2 wyznaczania zapisu U1 dla liczby dziesiętnej***

1) Jeśli liczba jest dodatnia, znajdujemy jej reprezentację w naturalnym kodzie binarnym i uzupełniamy bitami o wartości 0 do uzyskania zadanej liczby bitów wymaganej przez przyjęty format zapisu U1.
2) Jeśli liczba jest ujemna, wyznaczamy wartość 2n - 1 + liczba, gdzie n oznacza liczbę bitów w przyjętym formacie U1. Wartość tę kodujemy w naturalnym systemie dwójkowym i otrzymujemy kod U1 liczby wyjściowej.

*Przykład : Wyznaczyć 8-mio bitowy zapis U1 liczby dziesiętnej (-113).*
Obliczamy:

$$2^8 - 1 - 113 = 256 - 1 - 113 = 142$$

Otrzymany wynik kodujemy w systemie dwójkowym i otrzymujemy kod U1 liczby -113:

$$142_{(10)} = 10001110_{(2)}, \text{ czyli } (-113)_{(10)} = 10001110_{(U1)}$$

*Przykład : Wyznaczyć 16 bitowy zapis U1 liczby dziesiętnej (-4521).*
Obliczamy:

$$2^{16} - 1 - 4521 = 65536 - 1 - 4521 = 61014$$

Otrzymany wynik kodujemy w systemie dwójkowym i otrzymujemy kod U1 liczby -45:

$$61014^{(10)} = 1110111001010110^{(2)}, \text{ czyli } (-4521)^{(10)} = 11101110010101100^{(U1)}.$$

**Zakres n bitowej liczby w kodzie U1 wynosi $Z_{(U1)} = (-2^{n-1} + 1 , 2^{n-1} - 1)$**

### *12.2.2. Stałoprzecinkowy zapis U1.*

Kod U1 może również reprezentować liczby ułamkowe, jeśli dokonamy rozszerzenia cyfr na pozycje ułamkowe. W przypadku liczb dodatnich (gdy pozycja znakowa zawiera cyfrę 0) stosujemy poznane zasady wyznaczania wartości liczby.

Przykład : 0111,1101(U1) = 22 + 21 + 20 + 2-1 + 2-2 + 2-4 = 4 + 2 + 1 + 1/2 + 1/4 + 1/16 = 713/16

Jednak przy liczbach ujemnych musimy wprowadzić drobną modyfikację w stosunku do liczb całkowitych. Kod uzupełnień do 1 ma taką własność, iż wyraz kodowy zbudowany z samych jedynek jest zawsze równy 0:

**111...111 , 111...111 = 0**
n bitów całkowitych    m bitów ułamkowych

Wynika z tego, iż suma wszystkich wag liczby musi sprowadzać się do 0. Zatem waga bitu znakowego spełnia równanie: $waga_{\text{znakowa U1}} = -(max_{\text{części całkowitej}} + max_{\text{części ułamkowej}})$

Załóżmy, iż nasz format U1 zbudowany jest z n bitów całkowitych i m bitów ułamkowych
$max_{\text{części całkowitej}} = 2^{n-1} - 1$
$max_{\text{części ułamkowej}} = (2^{m-1}) / 2m$
Stąd : $waga_{\text{znakowa U1}} = -(2^{n-1} - 1 + (2^{m-1}) / 2m)$
gdy m = 0, wzór redukuje się do podanego wcześniej wzoru $-2^{n-1} + 1$ dla liczb całkowitych.

Przykład :

$1011{,}1011_{(U1)} \rightarrow n = 4 \rightarrow m = 4$, zatem waga znakowa jest równa $-(2^3 - 1 + (2^4 - 1) / 2^4) = -7^{15}/_{16}$.
$1011{,}1011_{(U1)} = -7^{15}/_{16} + 3 + 11/16 = -4^{4}/_{16}$

### *12.2.3. Dodawanie w U1*

Liczby U1 dodajemy zgodnie z poznanymi zasadami dodawania dwójkowego. Jednakże jeśli występuje przeniesienie poza bit znaku, to do wyniku należy dodać 1, aby otrzymać poprawny rezultat dodawania.

Przykład :

W podanych poniżej trzech przykładach przeniesienie poza bit znaku nie występuje, zatem wyniki operacji dodawania nie wymagają korekty i są poprawne.

2 + (-1) = 1

```
    1
    0 0 1 1
  + 0 0 1 0
  ---------
    0 1 0 1
```

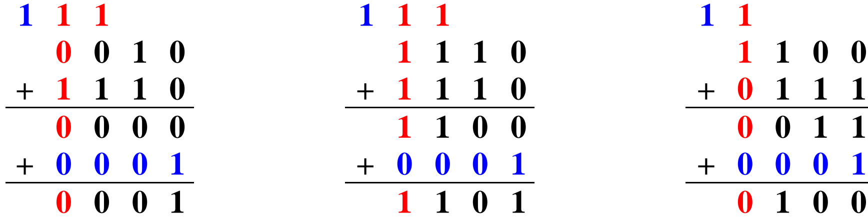
-1+(-1) = -2

```
    1
    0 0 1 1
  + 1 0 1 0
  ---------
    1 1 0 1
```

(-3) + 7 = 4

```
      1 1
    1 0 1 1
  + 0 0 1 1
  ---------
    1 1 1 0
```

W poniższych przykładach obserwujemy sytuację, gdy pojawia się przeniesienie poza bit znakowy (zapisaliśmy je w kolorze niebieskim). W takim przypadku wynik jest o 1 za mały i wymaga korekty. Ponieważ przeniesienie łatwo wykryć a układy zwiększające liczbę binarną o 1 są nieskomplikowane, dodawanie w kodzie U1 jest dosyć prosto realizowalne sprzętowo.



W przypadku dodawania liczb stałoprzecinkowych przeniesienie poza bit znaku dodawane jest do najmłodszej cyfry wyniku.

$2\,{}^{3}/_{16} + (-\,{}^{3}/_{16}) = 2$

```
  1 1 1
    0 0 1 0, 0 0 1 1
  + 1 1 1 1, 1 1 0 0
  ------------------
    0 0 0 1, 1 1 1 1
  + 0 0 0 0, 0 0 0 1
  ------------------
    0 0 1 0, 0 0 0 0
```

*12.2.4.Odejmowanie w U1.*

Odejmowanie realizujemy za pomocą dodawania liczby przeciwnej. Liczbę przeciwną tworzymy w kodzie U1 negując wszystkie bity zapisu liczby.

```
     5-6                              5+(-6)
                 operację 5 - 6            1
   0 1 0 1    zastępujemy operacją     0 1 0 1
 - 0 1 1 0        5 + (-6)           + 1 0 0 1
                                     ---------
                                       1 1 1 0
```

**UWAGA :** Przy dodawaniu dwóch liczb U1 może wystąpić nadmiar (niedomiar), gdy wynik dodawania nie mieści się w ustalonej formatem ilości bitów. Nadmiar (niedomiar) występuje tylko w przypadku dodawania liczb tego samego znaku. Zatem najprostszą metodą wykrycia nadmiaru (niedomiaru) jest sprawdzenie znaku wyniku. Jeśli znak wyniku operacji różni się od znaku argumentów, to wystąpił nadmiar (niedomiar).

## *12.3. Zapis uzupełnień do 2 – U2 (2C - Two's Complement )*

System zapisu U1 przedstawiony w poprzednim rozdziale umożliwiał w prosty sposób kodowanie liczb ujemnych. Jednakże wykonywanie operacji arytmetycznych wciąż wymaga dodatkowych założeń - przy dodawaniu do wyniku należało dodać przeniesienie poza bit znaku, aby otrzymać poprawny wynik.
Okazuje się, iż drobna modyfikacja systemu U1 pozwala ominąć te wady. W tym celu zmieniamy wagę bitu znakowego z $(-2^{n-1} + 1)$ na $(-2^{n-1})$, gdzie n oznacza ilość bitów w formacie kodu. Wagi stają się teraz jednorodne - bit znakowy posiada wagę ujemną, lecz o wartości bezwzględnej takiej samej jak w naturalnym kodzie dwójkowym. Nowy kod nosi nazwę uzupełnień do podstawy 2 lub w skrócie U2 (ang. 2C - Two's Complement).
Kod U2 jest powszechnie stosowany we wszystkich językach programowania do kodowania liczb całkowitych ze znakiem. Reguły arytmetyki NBC zachowane są bez zmian

| Wartości wag pozycji w zapisie U2 | | | | | | | |
|---|---|---|---|---|---|---|---|
| **waga** | $-2^{n-1}$ | $2^{n-2}$ | $2^{n-3}$ | ....... | $2^2$ | $2^1$ | $2^0$ |
| **cyfra** | $b_{n-1}$ | $b_{n-2}$ | $b_{n-3}$ | ....... | $b_2$ | $b_1$ | $b_0$ |

***Liczba jest dodatnia, gdy bit znaku ma wartość 0 - suma pozostałych wag tworzy zawsze liczbę dodatnią lub zero. Jeśli bit znaku przyjmie wartość 1, to liczba jest ujemna.***

Wartość liczby U2 obliczamy zgodnie z podanymi w rozdziale „Liczbowe systemy pozycyjne” zasadami - cyfry mnożymy przez wagi pozycji, na których się znajdują i dodajemy otrzymane iloczyny. Waga bitu znakowego jest ujemna.

$$b_{n-1}b_{n-2}b_{n-3}...b_2b_1b_0\ _{(U2)} = b_{n-1}(-2^{n-1}) + b_{n-2}2^{n-2} + b_{n-3}2^{n-3} + ... + b_2 2^2 + b_1 2^1 + b_0 2^0$$

gdzie
b - bit, cyfra dwójkowa 0 lub 1
n - liczba bitów w zapisie liczby

Ponieważ w systemach binarnych cyfry przyjmują tylko dwie wartości 0 lub 1, można rachunek znacznie uprościć sumując jedynie te wagi, dla których cyfra zapisu liczby ma wartość 1.

| 4 bitowy system U2 | | |
|---|---|---|
| **Kod U2** | **Przeliczenie** | **Wartość** |
| 0000 | 0 | 0 |
| 0001 | $2^0$ | 1 |
| 0010 | $2^1$ | 2 |
| 0011 | $2^1 + 2^0$ | 3 |
| 0100 | $2^2$ | 4 |
| 0101 | $2^2 + 2^0$ | 5 |
| 0110 | $2^2 + 2^1$ | 6 |
| 0111 | $2^2 + 2^1 + 2^0$ | 7 |
| 1000 | $(-2^3)$ | (-8) |
| 1001 | $(-2^3) + 2^0$ | (-7) |
| 1010 | $(-2^3) + 2^1$ | (-6) |
| 1011 | $(-2^3) + 2^1 + 2^0$ | (-5) |
| 1100 | $(-2^3) + 2^2$ | (-4) |
| 1101 | $(-2^3) + 2^2 + 2^0$ | (-3) |
| 1110 | $(-2^3) + 2^2 + 2^1$ | (-2) |
| 1111 | $(-2^3) + 2^2 + 2^1 + 2^0$ | (-1) |

W tabelce obok zostały zebrane wszystkie możliwe do utworzenia 4-ro bitowe liczby w zapisie U2. Należy zauważyć, że w przeciwieństwie do poprzednio opisanych systemów ZM i U1 w tym systemie wartość 0 ma tylko jedną reprezentację 0000, a liczb ujemnych jest o 1 więcej niż dodatnich (-8 .. -1, 1 .. 7).

Najstarszy bit określa znak liczby. Jeśli jest równy 0, liczba jest dodatnia i resztę zapisu możemy potraktować jak liczbę w naturalnym kodzie dwójkowym.

*Przykład :*

$01101011_{(U2)} = 64 + 32 + 8 + 2 + 1 = 107_{(10)}$.

Jeśli bit znaku ustawiony jest na 1, to liczba ma wartość ujemną. Bit znaku ma wagę $(-2^{n-1})$, gdzie n oznacza liczbę bitów w wybranym formacie U2. Reszta bitów jest zwykłą liczbą w naturalnym kodzie dwójkowym. Wagę bitu znakowego i wartość pozostałych bitów sumujemy otrzymując wartość liczby U2.

$11101011_{(U2)} = (-2^7) + 64 + 32 + 8 + 2 + 1 = -128 + 107 = (-21)_{(10)}$.

Zwróćmy uwagę, że postać ujemna liczby U2 nie jest już tak czytelna dla nas jak w przypadku kodów ZM (tylko zmiana bitu znaku) i U1 (negacja wszystkich bitów).

### *12.3.1. Liczba przeciwna do liczby w U2*

***Sposób 1 otrzymywania liczby przeciwnej do danej liczby U2.***

1. Dokonać negacji wszystkich bitów zapisu liczby U2.
2. Do wyniku dodać 1.

*Przykład : Wyznaczyć liczbę przeciwną w kodzie U2 do danej liczby $01101110_{(U2)}$.*

Dokonujemy negacji (zmianę na wartości przeciwne) wszystkich bitów liczby U2:
NOT 01101110 = 10010001
Do wyniku negacji dodajemy 1:

$$\begin{array}{r} 10010001 \\ +\quad 00000001 \\ \hline 10010010 \end{array}$$

Liczbą przeciwną do $01101110_{(U2)}$ jest $10010010_{(U2)}$.

Zwróćmy uwagę, że tym sposobem nie da się otrzymać liczby przeciwnej do najmniejszej liczby ujemnej (bit znaku ustawiony na 1, a wszystkie pozostałe bity równe 0). Na przykład dla 4 bitowego kodu U2 otrzymujemy:
$(-8)_{(10)} = 1000_{(U2)} \rightarrow$ NOT(1000) = 0111 $\rightarrow$ 0111 + 0001 = 1000, a to jest ta sama liczba wyjściowa.
Oczywistym wyjaśnieniem tego faktu jest to, iż najmniejsza w danym formacie U2 liczba ujemna nie posiada w tym formacie swojego odpowiednika po stronie dodatniej, gdyż suma wszystkich wag dodatnich jest o 1 mniejsza od modułu wagi ujemnej bitu znakowego.

***Sposób 2 otrzymywania liczby przeciwnej do danej liczby U2.***
1. Przejść do pierwszego od prawej strony bitu zapisu liczby.
2. Do wyniku przepisać kolejne bity 0, aż do napotkania bitu o wartości 1, który również przepisać.
3. Wszystkie pozostałe bity przepisać zmieniając ich wartość na przeciwną.

*Przykład : Znaleźć liczbę przeciwną w kodzie U2 do danej liczby*
$11001000101110101110100101000010000000_{(U2)}$.

Analizę liczby rozpoczynamy ostatniej cyfry zapisu liczby. Przesuwamy się w lewą stronę. Do wyniku przepisujemy wszystkie kolejne bity o wartości 0, aż do napotkania bitu 1.

| Liczba U2 | 11001000101110101110100101000010000000 |
|---|---|
| Liczba przeciwna U2 | 000000 |

Napotkany bit 1 również przepisujemy do wyniku bez zmian:

| Liczba U2 | 11001000101110101110100101000010000000 |
|---|---|
| Liczba przeciwna U2 | 1000000 |

Pozostałe bity przepisujemy zmieniając ich stan na przeciwny. To wszystko.

| Liczba U2 | 11001000101110101110100101000010000000 |
|---|---|
| Liczba przeciwna U2 | 00110111010001010001011010111111000000 |

Również drugi sposób zawodzi przy wyznaczaniu liczby przeciwnej do najmniejszej liczby ujemnej w danym formacie U2. Dlatego na wartość tę należy zwrócić szczególną uwagę.

### *12.3.2. Przeliczanie liczb dziesiętnych na zapis U2*

Dla liczb dodatnich nie ma problemu z przeliczaniem na kod U2. Wystarczy znaleźć reprezentację dwójkową danej wartości liczbowej (patrz : „Przeliczanie liczb dziesiętnych na zapis binarny”), a następnie uzupełnić ją bitami 0 do długości formatu kodu U2.

Przykład : Wyznaczyć 8-mio bitowy kod U2 dla liczby dziesiętnej $27_{(10)}$.
$27_{(10)} = 16 + 8 + 2 + 1 = 1\times2^4 + 1\times2^3 + 0\times2^2+1\times2^1+1\times2^0 = 11011_{(2)} = 00011011_{(U2)}$.

W przypadku wartości ujemnej mamy kilka możliwości postępowania, które opisujemy poniżej.

***Sposób 1 przeliczania ujemnej liczby dziesiętnej na zapis U2.***

Liczba ujemna musi mieć ustawiony na 1 bit znaku. Zatem nasze zadanie sprowadza się do znalezienia wartości pozostałych bitów. Bit znaku stoi na pozycji o wadze ($-2^{n-1}$), n - ilość bitów w formacie U2. Pozostałe bity zapisu liczby tworzą naturalny od dwójkowy. Wartość tego kodu musi być taka, aby po dodaniu jej do wagi pozycji znakowej otrzymać wartość kodowanej liczby. Zapiszmy to w formie równania:

*kodowana liczba* = waga bitu znakowego + wartość kodu pozostałych bitów

*kodowana liczba* = ($-2^{n-1}$) + wartość kodu pozostałych bitów

stąd :

*wartość kodu pozostałych bitów* = $2^{n-1}$ + kodowana liczba

Po wyznaczeniu wartości tego kodu tworzymy jego zapis w systemie dwójkowym, uzupełniamy w miarę potrzeby bitem 0 do długości formatu U2 - 1 i dodajemy bit znakowy 1. Konwersja jest gotowa.

*Przykład : Wyznaczyć 8-mio bitowy kod U2 dla liczby dziesiętnej* $(-45)_{(10)}$.

Wyznaczamy moduł wagi pozycji znakowej. Dla n = 8 $\rightarrow 2^{n-1} = 2^7 = 128$

wartość kodu pozostałych bitów = 128 + (-45) = 83 = $1010011_{(2)}$

Dodajemy bit znaku równy 1 i otrzymujemy:

$(-45)_{(10)} = 11010011_{(U2)}$.

***Sposób 2 przeliczania ujemnej liczby dziesiętnej na zapis U2.***

1. Wyznaczamy zapis dwójkowy liczby przeciwnej (czyli dodatniej).
2. Otrzymany kod dwójkowy uzupełniamy w miarę potrzeb do rozmiaru formatu U2.
3. Wyznaczamy liczbę przeciwną za pomocą jednej z opisanych wcześniej metod (rozdział 12.3.1)

*Przykład : Wyznaczyć 8-mio bitowy kod U2 dla liczby dziesiętnej* $(-45)_{(10)}$.

Wyznaczamy kod binarny liczby przeciwnej: $45_{(10)} = 101101_{(2)}$

Kod uzupełniamy dwoma bitami 0 do wymaganej długości 8 bitów: 00101101.

Wyznaczamy liczbę przeciwną wg drugiej metody: $00101101_{(U2)}$ : $11010011_{(U2)}$.

Stąd $(-45)_{(10)} = 11010011_{(U2)}$.

***Sposób 3 przeliczania ujemnej liczby dziesiętnej na zapis U2.***

Jeśli do liczby 2n (n - ilość bitów w formacie U2) dodamy przetwarzaną liczbę dziesiętną, to w wyniku otrzymamy wartość kodu dwójkowego równoważnego bitowo (tzn. o takiej samej postaci) kodowi U2 przetwarzanej liczby. Wynik dodawania wystarczy zapisać w postaci naturalnego kodu dwójkowego i konwersja jest zakończona.

*Przykład : Wyznaczyć 8-mio bitowy kod U2 dla liczby dziesiętnej* $(-45)_{(10)}$.

$2^8 + (-45) = 256 - 45 = 211 = 11010011_{(2)}$.

Stąd $(-45)_{(10)} = 11010011_{(U2)}$.

Jak widać, jest to metoda najszybsza z opisanych metod.

### *12.3.3. Stałoprzecinkowy zapis U2*

Zapis U2, podobnie jak opisane wcześniej zapisy ZM i U1, można rozszerzyć do zapisu stałoprzecinkowego dodając bity o wagach ułamkowych równych kolejnym, ujemnym potęgom podstawy 2. Zasada obliczania wartości liczby stałoprzecinkowej U2 w niczym nie różni się od poprzednio opisanych zasad (patrz : rozdział 11.5).

*Przykład :*

$0110{,}1011_{(U2)} = 2^2 + 2^1 + 2^{-1} + 2^{-3} + 2^{-4} = 4 + 2 + {}^1/_2 + {}^1/_8 + {}^1/_{16} = 6\ {}^{11}/_{16}$

$1101{,}0011_{(U2)} = (-2^3) + 2^2 + 2^0 + 2^{-3} + 2^{-4} = (-8) + 4 + 1 + {}^1/_8 + {}^1/_{16} = (-8) + 5\ {}^3/_{16} = -(2\ {}^{13}/_{16})$

Jeśli chcemy wyznaczyć liczbę przeciwną do danej liczby stałoprzecinkowej w kodzie U2, to stosując pierwszą z opisanych powyżej metod zamiast 1 do zanegowanych bitów dodajemy liczbę stałoprzecinkową z ustawionym na 1 bitem na najmłodszej pozycji.

*Przykład : Obliczamy liczbę przeciwną do $1101{,}1001_{(U2)} = (-2\ {}^7/_{16})$*

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| NOT | 1 | 1 | 0 | 1 | , | 1 | 0 | 0 | 1 | |
| | 0 | 0 | 1 | 0 | , | 0 | 1 | 1 | 0 | |
| + | 0 | 0 | 0 | 0 | , | 0 | 0 | 0 | 1 | |
| | 0 | 0 | 1 | 0 | , | 0 | 1 | 1 | 1 | $= 2\ {}^7/_{16}$ |

Wartość dziesiętną przeliczamy na zapis stałoprzecinkowy U2 wg następujących zasad:

- Jeśli liczba jest dodatnia, to stosujemy dowolną z opisanych wcześniej metod (rozdział 11.5.1) w celu wyznaczenia jej reprezentacji binarnej. Wynik musimy jednak uzupełnić bitami 0 do długości wybranego formatu U2.

*Przykład : Znaleźć zapis U2 liczby 3,125. Zapis U2 posiada 4 bity całkowite i 4 bity ułamkowe.*

$3_{(10)} = 11_{(2)}$ - część całkowita
$0{,}125_{(10)} = {}^1/_8 = 0{,}001_{(2)}$ - część ułamkowa

Łączymy obie części w całość i uzupełniamy odpowiednio bitami 0 do wymagań formatu U2:
$3{,}125_{(10)} = 11{,}001_{(2)} = 0011{,}0010_{(U2)}$.

- Jeśli liczba jest ujemna, to jej zapis U2 możemy znaleźć na kilka sposobów. Na przykład można obliczyć zapis binarny liczby przeciwnej do niej (czyli dodatniej), uzupełnić bitami 0, a następnie zamienić otrzymany od U2 na liczbę przeciwną. Jednakże proponuję bardzo prostą metodę, która bezpośrednio pozwala wyznaczyć kod binarny liczby U2. Jeśli zapis U2 posiada n cyfr całkowitych, to liczbę wyjściową sumujemy z wartością $2^n$. W wyniku otrzymujemy wartość naturalnego kodu dwójkowego, który bitowo jest równoważny kodowi U2 przeliczanej liczby.

*Przykład : Znaleźć zapis U2 liczby -2,125. Zapis U2 posiada 3 bity całkowite oraz 5 bitów ułamkowych.*

Ponieważ bitów całkowitych jest n=3, to liczbę -2,125 sumujemy z liczbą $2^3 = 8$:

$8 + (-2,125) = 5,875 = 5\,{}^{7}/_{8}$

Otrzymana wartość zamieniamy na liczbę w naturalnym kodzie binarnym, po czym uzupełniamy bitami 0 do specyfikacji formatu U2:

$5,875_{(10)} = 101,111_{(2)} = 101,11100$

Otrzymany kod jest stałoprzecinkowym kodem U2 liczby wyjściowej:

$-2,125_{(10)} = 101,11100_{(U2)}$.

### *12.3.4. Dodawanie i odejmowanie w U2.*

Liczby U2 dodajemy i odejmujemy według poznanych zasad dla naturalnego systemu dwójkowego (rozdział 11.6.1). Przeniesienia poza bit znaku ignorujemy.

Przykład:

3+3=6

| | | | | |
|---|---|---|---|---|
| | 0 | 0 | 1 | 1 |
| + | 0 | 0 | 1 | 1 |
| | 0 | 1 | 1 | 0 |

5+(-4)=1

| | | | | |
|---|---|---|---|---|
| | 0 | 1 | 0 | 1 |
| + | 1 | 1 | 0 | 0 |
| **1** | 0 | 0 | 0 | 1 |

5-4=1

| | | | | |
|---|---|---|---|---|
| | 0 | 1 | 0 | 1 |
| - | 0 | 1 | 0 | 0 |
| | 0 | 0 | 0 | 1 |

-7-(-6)= -1

| | | | | |
|---|---|---|---|---|
| | 1 | 0 | 0 | 1 |
| - | 1 | 0 | 1 | 0 |
| **1** | 1 | 1 | 1 | 1 |

### *12.3.5. Mnożenie w U2.*

Mnożenie liczb w kodzie U2 różni się nieco od standardowego mnożenia liczb binarnych. Przed wykonaniem tej operacji arytmetycznej musimy rozszerzyć znakowo obie mnożone liczby tak, aby ich długość (liczba bitów) wzrosła dwukrotnie (jeśli są różnej długości, to rozszerzamy znakowo względem dłuższej liczby). Rozszerzenie znakowe polega na powielaniu bitu znaku na wszystkie dodane bity. Np.:

**0**$111_{(U2)}$ = **0000 0**$111_{(U2)}$ - rozszerzyliśmy znakowo liczbę 4 bitową do 8 bitowej

**1**$011_{(U2)}$ = **1111 1**$011_{(U2)}$ - to samo dla liczby ujemnej.

Rozszerzenie znakowe nie zmienia wartości liczby w kodzie U2. Po wykonaniu rozszerzenia znakowego liczby mnożymy standardowo.

Przykład :

$-2 = 1110_{(U2)} = 1111\ 1110_{(U2)}$

$3 = 0011_{(U2)} = 0000\ 0011_{(U2)}$

(-2) x 3 = -6

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| x | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | **0** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| + | **1** | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| **1** | **0** | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

Wynik mnożenia może być liczbą o długości równej sumie długości mnożonych liczb. Dlatego bity wykraczające w naszym przykładzie poza 8 bitów ignorujemy. Pozostałe 8 bitów określa w kodzie U2 liczbę -6. Zatem rachunek zgadza się.

### *12.3.6. Dzielenie w U2.*

Najprostszym rozwiązaniem jest zapamiętanie znaków dzielonych liczb, zamiana ich na liczby dodatnie, dokonanie dzielenia dla liczb naturalnych, a następnie zmiana znaku wyniku, jeśli znaki dzielnej i dzielnika różnią się.

Przykład : podzielmy 6 przez -3:
$6 = 0110_{(U2)}$
$-3 = 1101_{(U2)}$ - zmieniamy na $3 = 0011_{(U2)}$
Dzielimy liczbę 0110 przez 0011

```
    10
 -------------
   0110: 0011
 - 0110
 ------
   0000
```

Otrzymaliśmy wynik 0010 (liczba 2). Ponieważ znaki dzielnej i dzielnika są różne, zmieniamy wartość na przeciwną: 1110 i ostatecznie otrzymujemy wynik $1110_{(U2)} = (-2)$.

Jeśli w trakcie dzielenia otrzymamy resztę, to musi ona mieć ten sam znak, co dzielna. Zbierzmy i podsumujmy reguły określania znaków wyniku i reszty z dzielenia w poniższej tabelce. Zawiera ona stan bitów znakowych.

| **Reguły znaków przy dzieleniu liczb całkowitych** | | | |
|---|---|---|---|
| **Dzielna** | **Dzielnik** | **Wynik** | **Reszta** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **1** |
| **1** | **1** | **0** | **1** |

# 13. Pozostałe kody binarne.

## *13.1. Kod BCD.*

Istnieje wiele przykładów urządzeń, w których zastosowanie czystego kodu dwójkowego jest nieekonomiczne z uwagi na ciągłą konieczność przeliczania liczb pomiędzy systemami dziesiętnym i dwójkowym. Są to różnego rodzaju liczniki, kasy sklepowe, kalkulatory, wagi itp. Dla nich opracowano specjalny kod zwany systemem dziesiętnym kodowanym dwójkowo - BCD (ang. Binary Coded Decimal).

| Kody cyfr dziesiętnych w systemie BCD | |
|---|---|
| **cyfra** | **kod BCD** |
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

Idea kodu jest bardzo prosta - dwójkowo zapisujemy nie wartość liczby lecz jej cyfry dziesiętne. Każda cyfra dziesiętna może być przedstawiona w postaci wartości w naturalnym kodzie binarnym. Do tego celu potrzebne są 4 bity. Kody poszczególnych cyfr przedstawiliśmy w tabelce obok.
Aby odczytać wartość liczby BCD wystarczy podzielić jej kod na grupy 4 bitowe. Każdą grupę zamieniamy zgodnie z tabelką na cyfrę dziesiętną i otrzymujemy zapis liczby w systemie dziesiętnym:

*Przykład :*

$10000100_{(BCD)} = 1000\ 0100 = 84_{(10)}$
$010101110010_{(BCD)} = 0101\ 0111\ 0010 = 572_{(10)}$
$0011011110001001_{(BCD)} = 0011\ 0111\ 1000\ 1001 = 3789_{(10)}$

W odwrotną stronę jest również prosto - każdą cyfrę dziesiętną zastępujemy 4 bitami z tabelki i otrzymujemy kod BCD:
$72398015_{(10)} = 0111\ 0010\ 0011\ 1001\ 1000\ 0000\ 0001\ 0101_{(BCD)}$

Jeśli bity liczby BCD zostaną przekonwertowane na system szesnastkowy, to otrzymane cyfry szesnastkowe odpowiadają dokładnie cyfrom dziesiętnym wartości liczby BCD. Z tej własności często korzystają programiści zapisując kod BCD właśnie w systemie szesnastkowym.
Kod BCD nie jest kodem efektywnym, ponieważ nie wykorzystuje wszystkich możliwych kombinacji bitów w słówkach kodowych - sprawdź to w tabelce. Wynika z tego oczywisty wniosek, iż niektóre słowa kodowe nie są dozwolone, gdyż nie reprezentują cyfr dziesiętnych:

*Przykład :*

$110111111010_{(BCD)} = 1101\ 1111\ 1010 = ???_{(10)}$

8 bitowa liczba BCD zawiera dwie cyfry dziesiętne, zatem może przyjąć wartości z zakresu od 00 do 99, co daje 100 słów kodowych. Tymczasem 8 bitów można ze sobą połączyć na 256 (28) sposobów. Zatem 156 słów kodowych nie będzie wykorzystanych przez kod BCD - to więcej niż połowa. Nadmiarowe słówka kodowe można wykorzystać do kontroli poprawności kodu BCD - jeśli otrzymamy zabronioną kombinację bitów, to od razu wiemy, że coś poszło źle.

Podchodząc do zagadnienia w sposób formalny, możemy wyprowadzić wzór, który na podstawie stanu bitów liczby BCD pozwoli nam obliczyć jej wartość dziesiętną. W tym celu rozważmy wagi pozycji 12-bitowej liczby BCD:
Waga i-tej pozycji w kodzie BCD ma wartość $10^{[i/4]} \times 2^{i \bmod 4}$. Wobec tego wzór na wartość n-bitowej liczby BCD jest następujący:

$$b_{n-1}...b_1b_0 = b_{n-1} \times 10^{[(n-1)/4]} \times 2^{(n-1) \bmod 4} + ... + b_1 \times 10^0 \times 2^1 + b_0 \times 10^0 \times 2^0$$

Wzór do najprostszych nie należy i chyba szybciej znajdziemy wartość liczby grupując bity i zamieniając je na cyfry dziesiętne.

### *13.2. Kod Gray'a.*

Wyobraźmy sobie, iż konstruujemy ramię robota. Ramię to ma zataczać kąt 32º (na potrzeby przykładu, w rzeczywistości może to być dowolny kąt). W przegubie ramienia umieszczamy czujnik, który odczytuje położenie ramienia i przekazuje kąt obrotu w postaci 5-bitowego naturalnego kodu binarnego.

00000 --- 0º
00001 --- 1º
00010 --- 2º
...
11110 --- 30º
11111 --- 31º

Na pierwszy rzut oka rozwiązanie to wydaje się całkiem sensowne. Mamy ramię. Mamy czujnik obrotu i mamy kod binarny, za pomocą którego czujnik przekazuje informację o położeniu tegoż ramienia. Dane te komputer sterujący może wykorzystywać do kierowania ruchami robota.

A teraz nadchodzi rzeczywistość. Bity z czujnika nie zmieniają się równocześnie - układ pomiarowy ma luzy. Z tego powodu w pewnych sytuacjach mogą pojawić się nieprawidłowe dane z czujnika. Na przykład wyobraźmy sobie, iż ramię obraca się z położenia 15º do położenia 16º:

01111(2) --- 15º - stan początkowy
11110(2) --- 31º - stan przejściowy, bity środkowe nie zdążyły się jeszcze zmienić
11010(2) --- 26º - stan przejściowy, jeszcze są bity niezmienione
10000(2) --- 16º - stan końcowy, ramię osiągnęło zadane położenie

Zwróć uwagę, iż w podanym przykładzie jeśli odczyt danych z czujnika nastąpi przed ustaleniem się stanów jego bitów (po zaniknięciu luzów w układzie pomiarowym), to odczytana wartość ma niewiele wspólnego z rzeczywistym położeniem ramienia robota. Jeśli program sterujący nie uwzględnia błędnych odczytów, może dojść do bardzo dziwnych rzeczy - np. ramię zacznie oscylować w jedną i w drugą stronę, ponieważ komputer sterujący sądzi, iż jest ono w innym położeniu niż powinno być.
Dlaczego tak się dzieje - po prostu zastosowaliśmy zły kod. Idealny kod pomiarowy powinien zmieniać tylko jeden bit przy przechodzeniu do następnej wartości - nasz zmienia tu wszystkie bity, stąd jeśli zmiana nie nastąpi synchronicznie, mogą pojawić się kłopoty. Taki kod istnieje i nosi nazwę kodu Gray'a - kolejne wyrazy kodu Gray'a różnią się od siebie stanem tylko jednego bitu.

Przy poprzednim kodzie niepewność odczytu mogła w niekorzystnych warunkach dotyczyć wszystkich bitów słowa kodowego. W przypadku kodu Gray'a niepewność ta dotyczy tylko 1 bitu, gdyż tylko jeden bit zmienia swój stan przy przejściu do następnego słowa kodowego. Zatem przykład może wyglądać następująco:

$01000_{(GRAY)}$ --- 15º - stan początkowy
$01000_{(GRAY)}$ --- 15º - stan pośredni, bity jeszcze się nie ustaliły
$11000_{(GRAY)}$ --- 16º - stan końcowy

W kodzie Gray'a słowa kodowe o wartości 15 (01000) i 16 (11000) różnią się tylko jednym bitem. Zatem nie dojdzie do dużych błędów odczytu położenia.

Tak więc znając już zastosowanie kodu Gray'a (a zastosowań ma bardzo dużo), przejdziemy do sposobu konstrukcji poszczególnych wyrazów. Podana niżej metoda tworzy tylko jeden z możliwych kodów Gray'a.

### *13.2.1. Wyznaczanie i-tego wyrazu n-bitowego kodu Gray'a.*

1. Zapisujemy numer wyrazu kodu Gray'a w naturalnym kodzie dwójkowym na zadanej liczbie bitów. Brakujące bity uzupełniamy bitem 0.
2. Pod spodem wypisujemy ten sam numer przesunięty w prawo o 1 bit. Najmniej znaczący bit odrzucamy. Na początku dopisujemy bit o wartości 0.
3. Nad odpowiadającymi sobie bitami wykonujemy operację logiczną XOR. Wynik jest wyrazem w kodzie Gray'a.

Przypominam, że operacja XOR to suma symetryczna. Jest to operacja dwuargumentowa.

| a | b | a XOR b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Zuważmy, że wynikiem sumy symetrycznej jest 1 tylko wtedy, gdy dokładnie jeden z argumentów ma wartość 1 i 0 w przeciwnym przypadku.

*Przykład :*

Kod dwójkowy 1011 -> 1011 XOR 0101 = 1110 Kod Gray'a
Kod dwójkowy 1111 -> 1111 XOR 0111 = 1000 Kod Gray'a

Przekształcenie kodu dwójkowego w kod Gray'a jest dosyć proste. Równie mało skomplikowana jest operacja odwrotna. Przeliczenia dokonujemy kopiując najstarszy bit kodu Gray'a do najstarszego bitu kodu binarnego. Pozostałe bity otrzymamy wykonując operację XOR na i-tym bicie kodu Gray'a i (i+1)-bicie wyrazu binarnego. Dla przykładu przeliczmy słowo w kodzie Gray'a 1110 na odpowiadające mu słowo dwójkowe:

Przeliczanie kodu Gray'a na naturalny kod dwójkowy

| Operacja | Opis |
|---|---|
| 1 1 1 0<br>1 ? ? ? | Przepisujemy najstarszy bit z kodu Gray'a do najstarszego bitu słowa dwójkowego. Bity te w obu kodach mają tę samą wartość. |
| 1 1 1 0<br>XOR 1 .<br>1 0 ? ? | Otrzymany bit umieszczamy na pozycji przesuniętej w prawo o jedno miejsce i wykonujemy operację XOR z bitem kodu Gray'a. W ten sposób otrzymujemy kolejne bity kodu dwójkowego. W tym przypadku dostajemy bit o wartości 0. |
| 1 1 1 0<br>XOR 0 .<br>1 0 1 ? | Identyczną operację wykonujemy z poprzednio otrzymanym bitem o wartości 0 otrzymując bit kodu dwójkowego o wartości 1. |
| 1 1 1 0<br>XOR 1<br>1 0 1 1 | Ostatnia operacja daje w wyniku słowo binarne 1011, które posłużyło do utworzenia słowa kodu Gray'a o wartości 1110. |

### *13.2.2.Rekurencyjny algorytm tworzenia wyrazów kodu Gray'a.*

Kod Gray'a jest kodem kombinatorycznym. Poszczególne pozycje bitów nie posiadają ustalonych wag, jak w przypadku opisanych poprzednio kodów binarnych. Wyrazy można tworzyć rekurencyjnie z wyrazów już wcześniej utworzonych wg następującego schematu:

Definiujemy działania na obiektach używanych przez algorytm:

- Jeśli L jest listą wyrazów binarnych, to L jest listą tych wyrazów wziętych w odwrotnej kolejności.
  Na przykład
  L = {00,01,11,10}
  L = {10,11,01,00} - kolejność odwrotna
- b+L jest nową listą utworzoną przez dołączenie bitu b na początek wszystkich wyrazów listy L.
  Na przykład
  L = {00,01,11,10} - lista wyrazów 2-bitowych
  0+L = {000,001,011,010} - lista wyrazów 3-bitowych
  1+L = {100,101,111,110} - lista wyrazów 3-bitowych
- $L_1 \oplus L_2$ jest listą powstałą przez złączenie tych dwóch list.
  Na przykład
  $L_1$ = {000,001,011,010} - lista 4-elementowa
  $L_2$ = {100,101,111,110} - lista 4-elementowa
  $L_1 \oplus L_2$ = {000,001,011,010,100,101,111,110} - lista 8-elementowa

Nowa lista wyrazów n-bitowego kodu Gray'a powstaje wg wzoru rekurencyjnego:
Dla n > 0
Jeśli n = 1, to $L_{GRAY}(1)$ = {0,1} w przeciwnym przypadku
$L_{GRAY}(n) = 0 + L_{GRAY}(n-1) \oplus 1 + L_{GRAY}(n-1)$
*Przykład : utworzymy 4-bitowy kod Gray'a wg opisanej metody rekurencyjnej.*

Rozpoczynamy od n=1. Dla tego przypadku metoda tworzy listę bezpośrednią:
$L_{GRAY}(1)$ = {0,1}
Teraz tworzymy listę wyrazów kodu Gray'a dla n=2 wykorzystując listę utworzoną dla n=1.
Wyrazy tej listy zaznaczyliśmy kolorem czerwonym.
$L_{GRAY}(2) = 0 + L_{GRAY}(1) \oplus 1 + L_{GRAY}(1)$
$0 + L_{GRAY}(1)$ = {00,01}
$1 + L_{GRAY}(1)$ = {11,10}
$L_{GRAY}(2)$ = {00,01} ⊕ {11,10}
$L_{GRAY}(2)$ = {00,01,11,10}
Wyznaczamy listę 3-bitowego kodu Gray'a na podstawie listy 2-bitowego kodu:
$L_{GRAY}(3) = 0 + L_{GRAY}(2) \oplus 1 + L_{GRAY}(2)$
$0 + L_{GRAY}(2)$ = {000,001,011,010}
$1 + L_{GRAY}(2)$ = {110,111,101,100}
$L_{GRAY}(3)$ = {000,001,011,010} ⊕ {110,111,101,100}
$L_{GRAY}(3)$ = {000,001,011,010,110,111,101,100}
Podobnie postępujemy dla listy 4-bitowego kodu Gray'a:
$L_{GRAY}(4) = 0 + L_{GRAY}(3) \oplus 1 + L_{GRAY}(3)$
$0 + L_{GRAY}(3)$ = {0000,0001,0011,0010,0110,0111,0101,0100}
$1 + L_{GRAY}(3)$ = {1100,1101,1111,1110,1010,1011,1001,1000}

$L_{GRAY}(4) = \{0000,0001,0011,0010,0110,0111,0101,0100\} \oplus$
$\{1100,1101,1111,1110,1010,1011,1001,1000\}$
$L_{GRAY}(4) =$
$\{0000,0001,0011,0010,0110,0111,0101,0100,1100,1101,1111,1110,1010,1011,1001,1000\}$

*Przykładowy program.*

```
{ Generowanie wszystkich wyrazow kodu Graya }
{         o zadanej liczbie bitow           }
{-------------------------------------------}
{ (C)2005 mgr Jerzy Walaszek                }
{  I Liceum Ogolnokszta lcace               }
{  im. K. Brodzinskiego w Tarnowie          }
{-------------------------------------------}

program KodGraya;
Uses Crt;

{ W tablicy WyrazyGraya tworzone beda kolejne  }
{ wyrazy kodowe. Tablica ta musi posiadac tyle }
{ elementow, ile jest wszystkich wyrazow kodu. }

var
  WyrazyGraya : array[0..30000] of word;

{ Funkcja oblicza potege liczby 2  }
{ -------------------------------- }
function Pot2(n : LongInt) : LongInt;
var
  p : LongInt;
begin
  p := 1;
  while n > 0 do
  begin
    p := p + p;
    n:=n-1;
  end;
  Pot2 := p;
end;

{ Rekurencyjna procedura generacji wyrazow kodu Graya }
{---------------------------------------------------- }
procedure Gray(n : LongInt);
var
  i,p : LongInt;
begin
  if n = 1 then
  begin
    WyrazyGraya[0] := 0;
    WyrazyGraya[1] := 1;
  end
  else
  begin
    Gray(n - 1); { wyznaczamy poprzednie wyrazy }
    p := Pot2(n - 1);
    for i := p  to p + p - 1 do
      WyrazyGraya[i] := p + WyrazyGraya[p + p - i - 1];
  end;
```

```
end;

{ Procedura wyswietlajaca zawartosc tablicy WyrazyGraya }
{------------------------------------------------------ }
procedure Pisz(n : LongInt);
var
  i,j,kg : LongInt;
  s      : string;
begin
  for i := 0 to Pot2(n) - 1 do
  begin
    s  := '';
    kg := WyrazyGraya[i];
    for j := 1 to n do
    begin
      s  := char(48 + kg mod 2) + s;
      kg := kg div 2;
    end;
    writeln(s);
  end;
end;

Var
   n : LongInt;
begin
  ClrScr;
  writeln('Generacja kodu Gray''a');
  writeln('---------------------');
  writeln('(C)2005 J.Walaszek');
  writeln('      I LO w Tarnowie');
  writeln;
  write('Ile bitow (1..16) ? ');
  readln(n);
  writeln;
  if not(n in [1..16]) then
    writeln('Niepoprawna liczba bitow n!')
  else
  begin
    Gray(n); Pisz(n);
  end;
  Writeln;Writeln;
  Writeln (' Nacisnij dowolny klawisz');
  Repeat Until KeyPressed;
End.
```

# 14. Szyfrowanie danych.

## *14.1. Steganografia*

Terminem steganografia określa się ukrywanie przekazywanych wiadomości. Pierwsze wzmianki na ten temat znajdujemy w historii zmagań między Grecją a Persją w V w. p.n.e., opisanych przez sławnego historyka Starożytności, Herodota. Przybierała ona różne formy, np. wiadomość zapisywano na cienkim jedwabiu, robiono z niego kulkę i po oblaniu jej woskiem goniec ją połykał. Inną formą steganografii jest stosowanie sympatycznego atramentu - tekst nim zapisany znika, ale pojawia się na ogół po podgrzaniu papieru.

Steganografia ma podstawową wadę - przechwycenie wiadomości jest równoznaczne z jej ujawnieniem. Od pojawienia się łączności drogą bezprzewodową za pomocą fal radiowych zaczęła mieć marginalne znaczenie w systemach łączności. Obecnie czasem odżywa dzięki możliwościom miniaturyzacji możliwe jest ukrycie nawet znacznej wiadomości, dodatkowo wcześniej zaszyfrowanej, np. w... kropce, umieszczonej .w innym tekście.

## *14.2. Kryptografia*

Terminem kryptografia określa się utajnianie znaczenia wiadomości. W praktyce, dla zwiększenia bezpieczeństwa stosuje się steganografię i kryptografię jednocześnie, czyli ukrywa się przesyłanie utajnionych wiadomości.

Szyfr, to ustalony sposób utajniania (czyli szyfrowania) znaczenia wiadomości. Wiadomość, która utajniamy określa się mianem tekstu jawnego, a jego zaszyfrowaną wersję - kryptogramem. Metody ukrywania znaczenia tekstu polegają na zastąpieniu go innym tekstem, z którego trudno domyśleć się znaczenia tekstu oryginalnego. Można to zrobić na dwa sposoby: przestawiając tylko znaki albo zastępując je innymi znakami. Ten pierwszy sposób nazywa się szyfrowaniem przez przestawianie (jest to tzw. szyfr przestawieniowy), a ten drugi - przez podstawianie (jest to tzw. szyfr podstawieniowy).

Łamaniem szyfrów, czyli odczytywaniem utajnionych wiadomości bez znajomości wszystkich elementów sposobu szyfrowania, zajmuje się kryptoanaliza. Kryptografia i kryptoanaliza to dwa główne działy kryptologii, nauki o utajnionej komunikacji.

Szyfrowanie danych znane już było od starożytności. To właśnie stąd pochodzi jedna z najstarszych metod zwana szyfrem Cezara. Polega ona na tym, że każdy znak tekstu jawnego jest przesuwany o trzy pozycje w prawo. Innym przykładem tego rodzaju algorytmu jest prosty programik dostarczany wraz z systemem UNIX - ROT13. Różni się on jedynie tym, że przesunięcie jest równe 13, a nie 3. Daje to tą dogodność, że ten sam program może szyfrować i odszyfrowywać daną wiadomość - alfabet ma 26 znaków, więc po dwukrotnej rotacji dostaniemy tekst początkowy. Metoda taka należy do kategorii prostych szyfrów podstawieniowych i jest łatwa do złamania - szyfry podstawieniowe nie zmieniają właściwości statystycznych występowania danego znaku. Oznacza to, że badając częstotliwość występowania danego znaku w języku, możemy z dużym prawdopodobieństwem go odtworzyć.

Trochę lepsze wyniki przedstawiają homofoniczne szyfry podstawieniowe, które jednemu znakowi tekstu jawnego przyporządkowują kilka znaków. Te były już używane w 1401r. przez księcia Mantui. Kolejną modyfikacją wprowadzoną do szyfrów podstawieniowych było wykorzystanie kilku alfabetów. Zmiana alfabetu może występować np. wraz z pozycją znaku w szyfrowanym tekście. Algorytm taki został wprowadzony przez Leona Battistę w 1568r. Był on również wykorzystywany przez armię Konfederatów podczas wojny domowej w USA.

Sposobem na utrudnienie analizy częstotliwości jest szyfrowanie poprzez podstawianie bloków liter określonej długości zamiast pojedynczych liter. Zauważymy wówczas zmniejszenie dysproporcji w zakresie częstotliwości występowania poszczególnych konfiguracji liter w bloku. Takim szyfrem jest np. Playfair, wprowadzony w 1854r. i stosowany przez Brytyjczyków w czasie I wojny światowej.
Oprócz szyfrów podstawieniowych znamy z historii również szyfry przestawieniowe - to takie, w których wszystkie znaki tekstu jawnego pojawiają się w tekście zaszyfrowanym, lecz w innej kolejności. Algorytm tego rodzaju o nazwie "ADFGVX" był stosowany przez Niemców podczas II wojny światowej. Mimo swojej złożoności został on złamany przez francuskiego kryptoanalityka George'a Painvina.
Ponieważ szyfry przestawieniowe są uciążliwe w stosowaniu, w latach dwudziestych naszego stulecia skonstruowano wiele maszyn, które miały zautomatyzować proces szyfrowania. Najbardziej znaną maszyną tego typu jest Enigma, która była stosowana przez Niemców podczas II wojny światowej. Szyfr Enigmy został złamany jeszcze podczas wojny przez trzech polskich matematyków: Mariana Rejewskiego, Henryka Zygalskiego i Jerzego Różyckiego.

Współcześnie w kryptografii stosuje się dwa rodzaje systemów szyfrowych. Pierwszy nosi nazwę kryptografii klucza tajnego (private key cryptography) lub też kryptografii symetrycznej (symmetric cryptography). W systemie tego typu do szyfrowania i deszyfrowania wiadomości wykorzystywany jest ten sam klucz. Oczywista staje się więc potrzeba utrzymania tego klucza w ścisłej tajemnicy. W odniesieniu do środowiska sieci komputerowych tego typu wymagania stają się dość uciążliwe. Na korzyść kryptografii klucza symetrycznego przemawia jednak szybkość działania zastosowanych tu algorytmów, które były projektowane specjalnie pod tym kątem oraz dużej liczby możliwych kluczy. Pierwszym rozpowszechnionym na szeroką skalę systemem klucza tajnego jest **DES** (Data Encryption Standard). Został on opracowany przez firmę IBM w latach 70-tych na bazie algorytmu Lucifer Horsta Feistela. W 1977 Narodowe Biuro Standardów USA (ANSI) zaakceptowało ten algorytm jako standard szyfrowania danych nie utajnionych przez agencje rządowe. W dwadzieścia lat później szyfr ten został złamany. Stało się to dla klucza 52-bitowego dającego w sumie ponad 72 kwadryliony kombinacji (wg specjalistów aby mówić o bezpieczeństwie, trzeba używać kluczy o długości 128 bitów, a najlepiej 1024 bitów). Rocke Verser - programista z Loveland w Kolorado, który nadzorował prace nad łamaniem szyfru w grupie nazwanej DESCHALL - użył najprostszej z możliwych metod - ataku siłowego - tzn. przetestował każdy możliwy klucz. Grupa miała dużo szczęścia, ponieważ na rozwiązanie udało im się natrafić po pokonaniu ok. 25 procent możliwych kombinacji (atak ten trwał od lutego 97 do lipca 97).

Inne powszechnie używane algorytmy kryptografii klucza symetrycznego to:

- DESX - prosta modyfikacja algorytmu DES
- Blowfish - algorytm wynaleziony przez Bruce'a Schneiera. Wykorzystywano go wielokrotnie w różnych aplikacjach. Nie są znane żadne ataki przeciwko niemu. Blowfish jest używany w wielu popularnych programach kryptograficznych, np. Nautilius i PGPfone
- IDEA (International Data Encryption Algorithm) - opracowany w Zurychu przez Jamesa L. Masseya i Xuejie Lai i opublikowany w 1990 roku. IDEA używa 128-bitowego klucza i uważany jest za algorytm mocny (tzn. dotychczas nie złamany). Wykorzystywany jest m.in. przez program PGP do szyfrowania danych oraz wiadomości przesyłanych pocztą elektroniczną. Jego publiczne wykorzystanie zostało poważnie ograniczone ze względu na uzyskanie nań kilku patentów.
- SAFER - algorytm rozwinięty przez J.L. Masseya (jednego z współtwórców IDEA). Twierdzi się że zapewnia bezpieczne szyfrowanie przy dość) szybkiej implementacji nawet na 8 bitowych procesorach. Dostępne są dwie odmiany, jedna dla kluczy 64 bitowych, a druga dla 128 bitowych.

- RC2 (Rivers Cipher 2) - opracowany przez Ronalda Rivesta. Stanowił tajemnicę handlową firmy RSA Data Security do 1996 roku, kiedy to został ujawniony w anonimowym liście przesłanym do jednej z grup dyskusyjnych. Algorytm ten jest mocny, jednak istnieje grupa kluczy mniej odpornych na złamanie. Dostępny jest w różnych wersjach umożliwiających stosowanie klucza o długości od 1 do 2048 bitów.
- RC4 (Rivers Cipher 4) - opracowany przez Ronalda Rivesta. Podobnie jak RC2, stanowił tajemnicę do czasu ujawnienia w 1994 roku w anonimowym liście przesłanym do jednej z grup dyskusyjnych. Jest również algorytmem mocnym, umożliwiającym stosowanie klucza o długości od 1 do 2048 bitów.
- Skipjack - algorytm zaproponowany przez NIST do użytku w szyfrujących układach elektronicznych Clipper i Capstone. Układy te wyposażono jednak w mechanizm umożliwiający odpowiednim służbom rozkodowanie informacji, teoretycznie po uzyskaniu zezwolenia sądowego.

Drugi z kolei system, o wiele bardziej praktyczny, nazywany jest kryptosystemem klucza publicznego (public key cryptography) lub też kryptografią asymetryczną (asymmetric cryptography). W przeciwieństwie do metody klucza tajnego komunikujące się strony używają dwu różnych kluczy - jednego do zaszyfrowania przesyłki (klucz publiczny lub jawny), drugiego do jej rozkodowania (klucz tajny lub prywatny). Kiedy ktoś chce zaszyfrować swoją przesyłkę stosuje klucz publiczny adresata. W takim przypadku wiadomość odszyfrować może drugi klucz z pary - klucz tajny znany jedynie właściwemu odbiorcy przesyłki.
Najbardziej znanymi algorytmami klucza publicznego są:

- RSA (nazwa pochodzi od pierwszych liter nazwisk autorów) - opracowany przez Ronalda Rivesta, Adi Shamira i Leonarda Adelmana na uniwersytecie MIT. Siła algorytmu bazuje na złożoności problemu rozkładu dużych liczb naturalnych na czynniki pierwsze. RSA jest opatentowany w USA i korzystający z tego systemu muszą wnosić) opłaty licencyjne. Nie dotyczy to jednak rynku europejskiego, gdyż algorytm został opublikowany przed jego opatentowaniem w USA. Niewątpliwie algorytm ten jest najbardziej rozpowszechniony, jednocześnie jest uznawany za najbardziej odporny na ataki siłowe.
- El Gamal - bazuje na skomplikowanych obliczeniach logarytmów dyskretnych.
- Diffie-Hellman (nazwa pochodzi od nazwisk autorów) - opracowany przez Whitfielda Diffie i Martina Hellmana. Algorytm nie może być bezpośrednio stosowany do szyfrowania danych. Pozwala raczej na wyznaczenie przez nadawcę i odbiorcę jednego, tajnego klucza do szyfrowania bez konieczności wcześniejszej wymiany jakichkolwiek poufnych informacji. Tak wyznaczony klucz może być później wykorzystany do szyfrowania danych przy pomocy któregoś z algorytmów systemu klucza tajnego.

Większość z wymienionych powyżej metod szyfrowania znajduje zastosowanie w życiu codziennym. Przykładem może być technologia SSL (Secure Sockets Layer), opracowana przez Netscape Communications. SSL umożliwia szyfrowanie i uwierzytelnianie przekazywanych informacji oraz ustalanie prawdziwej tożsamości komunikujących się serwerów i przeglądarek WWW. Wersja 3 specyfikacji SSL przewiduje stosowanie algorytmów szyfrowania: DES, IDEA, RC2 i RC4, algorytmy RSA i DSS do tworzenia podpisów cyfrowych, natomiast w procesie negocjacji kluczy szyfrowania - stosowanie algorytmu Diffego-Hellmana. Zabezpieczenie przez SSL strony dokumentu hipertekstowego posiadają odmienny format adresu URL: "https://".

## *14.3. Szyfrowanie przez przestawianie.*

Przestawienie liter w tekście wiadomości jest jednym z najprostszych sposobów zmiany znaczenia wiadomości. Przestawiając litery w słowie możemy otrzymać inne słowo, zwane jego anagramem. A zatem taki sposób szyfrowania dłuższego tekstu jest uogólnieniem anagramu. Jeśli w danym tekście możemy dowolnie przestawiać litery, to otrzymujemy olbrzymią liczbę możliwych kryptogramów.

Ogólny wzór na ilość kombinacji słowa n-literowego wynosi n! . Najszybsze obecnie komputery są w stanie wykonać w ciągu sekundy około $10^{12}$ operacji tak więc aby odczytać kryptogram, będący anagramem tekstu jawnego, należałoby wziąć pod uwagę wszystkie możliwe przestawienia liter co zajmie dosyć długi okres czasu. Bez pewnych reguł przestawiania będzie to szyfr tak samo trudny dla osoby przechwytującej kryptogram, jak i dla odbiorcy zaszyfrowanej wiadomości.

Szyfr ten jest zaliczany do szyfrów prostych (transposition cipher). Polega on na przestawianiu kolejnych liter lub par liter w określonym schemacie. Zasadą takiego szyfru jest to , że zawsze pierwsza litera pozostaje nie zmieniona .

Przykład :
tekst jawny : AleksanderKwasniewski
umowa : zamianiamy miejscami litery znajdujące się obok siebie
tekst tajny : AelsknaedKrawnseiswik

Możemy zamieniać także grupy liter powiedzmy pary , albo zamieniać miejscami ostatnią literę z pierwszą i tak dalej - dowolność tego systemu kodowania jest bardzo duża.

Przykład :
tekst jawny : AleksanderKwasniewski
umowa : zamieniamy miescami pary liter znajdujące się obok siebie
tekst tajny : AksledeanwarKiesnkiws

*Ćwiczenie : Poniższy kryptogram został otrzymany za pomocą przestawienia każdych dwóch kolejnych liter w tekście jawnym poczynając od drugiej litery (zaniedbano odstępy miedzy słowami).*

**WAHITILEKODNIBGSEITNSTOIHGN**

*Odczytaj tekst jawny.*

***Specyfikacja algorytmu.***

*Dane wejściowe :*

szyfr - łańcuch znaków zawierający zaszyfrowany tekst
pozycja - pozycja w łańcuchu, od której zaczynamy przestawiać litery, pozycja ∈ N, pozycja >1
ilosc - ilosc przestawionych naraz znaków , ilosc ∈ N, ilosc > 1

*Dane wyjściowe :*

tekst - łańcuch znaków zawierający rozszyfrowany tekst

*Dane pomocnicze :*
wycinek_1,
wycinek_2 - lancuch znakowe do przechowywania grup znakow
i - zmienna licznikowa

***Lista kroków :***

Krok 1 : Podaj szyfr; Podaj pozycja; Podaj ilosc;
Krok 2 : Jesli pozycja > 1 Then tekst := Copy (szyfr,1,pozycja-1);
Krok 3 : i:= 0;
Krok 4 : Wykonuj kroki 5 , 6 i 7 dopóki i < od dlugości szyfru
Krok 5 : wycinek_1 := Copy (szyfr,pozycja+i,ilosc);
wycinek_2 := Copy (szyfr,pozycja+i+ilosc,ilosc);
Krok 6 : i := i+2*ilosc;
Krok 7 : tekst := tekst + wycinek_2+wycinek_1;
Krok 8 : Wyświetl tekst i zakończ algorytm

***Przykładowy program***

```
Program Szyfr_przestawieniowy;
Uses Crt;
Var szyfr,tekst,wycinek_1,wycinek_2 : String;
   pozycja,ilosc,i                  : Integer;
Begin
   ClrScr;
   Writeln;
   Writeln ('---------------------------------------------');
   Writeln (' Rozszyfowanie tekstu metoda przestawieniowa ');
   Writeln ('---------------------------------------------');
   Writeln;
   Write (' Podaj zaszyfrowany tekst : ');
   Readln (szyfr);
   Write (' Podaj pozycje, od ktorej tekst zostal zamieniony : ');
   Readln (pozycja);
   Write (' Podaj ile znak˘w w grupie zostalo przestawionych : ');
   Readln (ilosc);
   If pozycja > 1 Then tekst := Copy (szyfr,1,pozycja-1);
   i:= 0;
   While i < Length (szyfr) do
   Begin
      wycinek_1 := Copy (szyfr,pozycja+i,ilosc);
      wycinek_2 := Copy (szyfr,pozycja+i+ilosc,ilosc);
      i := i+2*ilosc;
      tekst := tekst + wycinek_2+wycinek_1;
   End;
   Writeln;
   Writeln (tekst);
   Repeat Until KeyPressed;
End.
```

Bardzo prosty szyfr przestawieniowy otrzymujemy stosując **metodę płotu**. W tej metodzie, najpierw kolejne litery tekstu jawnego są zapisywane na zmianę w dwóch rzędach, a następnie za kryptogram przyjmuje się ciąg kolejnych liter z pierwszego rzędu, po którym następuje ciąg

kolejnych liter z drugiego rzędu. Na przykład, metoda płotu zastosowana do słowa szyfrowanie, daje następujący kryptogram:

s   y   r   w   n   e
  z   f   o   a   i

**SYRWNEZFOAI**

„Płot" w tej metodzie szyfrowania może się składać z więcej niż dwóch rzędów, np. kryptogram powyższej wiadomości jawnej, otrzymany z użyciem trzech rzędów w „płocie", ma postać: SFWIZRAEYON

W przedstawionej metodzie szyfrowania można wyróżnić dwa elementy :

- ogólny algorytm szyfrowania, polegający na ustawianiu koleinych liter tekstu jawnego w pewnej liczbie rzędów i zwijaniu kolejno tych rzędów w tekst zaszyfrowany;
- klucz, którym jest liczba rzędów w płocie - dzięki niemu ten ogólny algorytm staje się konkretną procedurą szyfrowania.

Metoda szyfrowania z użyciem płotu nie jest najlepszym przykładem bezpiecznego klucza szyfrującego, gdyż możliwych jest w niej niewiele kluczy i wszystkie łatwo można sprawdzić.

Przedstawione dalej metody szyfrowania mają podobny charakter. Najistotniejszym elementem sposobu szyfrowania jest klucz, a nie algorytm, który na ogół jest powszechnie znany. Jest to podstawowe założenie kryptologii: **bezpieczeństwo systemu szyfrowania jest oparte na kluczach, a nie na sposobie działania.**

**Poziom bezpieczeństwa:** Bezpieczeństwo nie jest zapewnione

**Metody kryptoanalizy:** Analiza statystyczna tekstu.

## *14.4. Szyfrowanie przez podstawianie.*

W kryptografii najpowszechniej stosuje się metody podstawieniowe, które polegają na zamianie liter innymi literami. Kolejne litery, tworzące wiadomość, nie zmieniają więc swojego miejsca, ale zmieniają swoje znaczenie.

Jednym z najstarszych szyfrów podstawieniowych jest **szyfr Cezara**, pochodzący z I w. p.n.e. Polega on na zastąpieniu każdej litery tekstu jawnego literą położoną w alfabecie o trzy miejsca dalej.
W przypadku szyfrów podstawieniowych mówimy o alfabecie jawnym, w którym są zapisywane wiadomości, i o alfabecie szyfrowym, w którym są zapisywane utajniane wiadomości. W przypadku szyfru Cezara dla alfabetu języka polskiego mamy:

| alfabet jawny | a | ą | b | c | ć | d | e | ę | f | g | h | i | j | k | l | ł | m | n | ń | o | ó | p | q | r | s | ś | t | u | v | w | x | y | z | ź | ż |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| alfabet szyfrowy | c | ć | d | e | ę | f | g | h | i | j | k | l | ł | m | n | ń | o | ó | p | q | r | s | ś | t | u | v | w | x | y | z | ź | ż | a | ą | b |

Deszyfracja, jest operacją odwrotną do szyfrowania. Szyfrowanie o takiej własności nazywamy **szyfrowaniem z kluczem symetrycznym**, gdyż znajomość klucza nadawcy wystarcza do określenia klucza odbiorcy. Niestety, tak zaszyfrowany tekst może rozszyfrować osoba nieuprawniona po przechwyceniu zaszyfrowanej wiadomości wiedząc tylko, że nadawca zastosował szyfr Cezara. Klasyczny szyfr Cezara można uogólnić, dopuszczając jako alfabet szyfrowy przesunięcie alfabetu jawnego o dowolną liczbę znaków - mamy więc 34 możliwe alfabety. To jednak nadal niewielkie utrudnienie i nawet mało wprawiony kryptoanalityk po przechwyceniu kryptogramu zaszyfrowanego uogólnionym szyfrem Cezara łatwo określi, jaki był alfabet szyfrowy.

**Poziom bezpieczeństwa:** szyfr nie zapewnia bezpieczeństwa

**Metody kryptoanalizy:** analiza częstości występowania poszczególnych liter

Dla zainteresowanych, poniższa tabela przedstawia częstości (%) występowania liter w polskich tekstach oraz tzw. kody Huffmana (stosowane przy kompresji danych).

| Litera | Częstość | Kod | Litera | Częstość | Kod |
|---|---|---|---|---|---|
| a | 8,71 | 1111 | m | 2,64 | 01011 |
| ą | 0,86 | 000000 | n | 5,60 | 0110 |
| b | 1,29 | 010101 | ń | 0,20 | 000001101 |
| c | 3,91 | 11001 | o | 7,90 | 1101 |
| ć | 0,48 | 0000010 | ó | 1,25 | 010100 |
| d | 3,45 | 10011 | p | 3,01 | 01111 |
| e | 7,64 | 1011 | r | 4,63 | 0010 |
| ę | 1,18 | 010010 | s | 4,64 | 0011 |
| f | 0,38 | 00000111 | ś | 0,78 | 001110111 |
| g | 1,42 | 011100 | t | 3,63 | 10100 |
| h | 1,22 | 010011 | u | 1,92 | 101010 |
| i | 8,48 | 1110 | w | 4,62 | 0001 |
| j | 2,38 | 01000 | y | 3,88 | 11000 |
| k | 3,10 | 10010 | z | 5,95 | 1000 |
| l | 2,09 | 00001 | ż | 0,71 | 0111010 |
| ł | 1,95 | 101011 | ź | 0,10 | 000001100 |

Należy jeszcze zauważyć, że częstotliwość występowania spacji jest co najmniej dwukrotnie większa od częstotliwości występowania najczęstszej samogłoski. Podczas szyfrowania należy więc opuszczać znaki spacji, aby podział tekstu na pojedyncze słowa nie ułatwił procesu złamania szyfru.

W Internecie stosuje się sposób szyfrowania oznaczony przez ROT13, który polega na zastępowaniu każdej litery tekstu literą, która znajduje się w alfabecie o 13 pozycji dalej (litery ze znakami diakrytycznymi i inne znaki są pozostawiane bez zmian). Ten sposób szyfrowania jest zalecany w przypadkach tych wiadomości (lub ich fragmentów), które nie powinny być czytelne na pierwszy rzut oka, gdy jest to rozwiązanie zagadki lub krzyżówki, lub zakończenie filmu lub książki. Program pocztowy Netscape Messenger zawiera funkcję dekodowania wiadomości zaszyfrowanych metodą ROT 13.
W poniższym przykładowym programie wykorzystamy kilka własności.

Znaki są przechowywane w pamięci komputera w postaci liczb zwanych kodami znaków. Każda litera ma przyporządkowany sobie numer. W tabeli ponizej przedstawiono kody liter od A do Z w kodzie ASCII (American Standard Code for Information Interchange - Amerykański, Standardowy Kod dla Wymiany Informacji).

| Litera | A | B | C | D | E | F | G | H | I | J | L | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Kod ASCII | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |

Klucz szyfrujący jest informacją, która pozwala zaszyfrować dane lub odczytać dane zaszyfrowane. W przypadku kodu Cezara kluczem jest wartość przesunięcia w alfabecie literek kodu względem literek tekstu jawnego. Przy szyfrowaniu przesunięcie dodajemy do kodu znaku. Przy rozszyfrowywaniu przesunięcie odejmujemy od kodu znaku.

Problem pojawia się przy literach skrajnych, gdy dodanie przesunięcia daje kod większy od 90 lub odjęcie przesunięcia daje kod mniejszy od 65. Jeśli umówimy się, iż alfabet tworzy pierścień zamknięty, to po literze Z następuje A. Literę A poprzedza Z. Jeśli w wyniku dodania przesunięcia otrzymamy wartość większą od 90 (np. 92), to od tego wyniku należy odjąć 26, aby otrzymać poprawny kod szyfru (92 - 26 = 66, czyli litera B). Jeśli z kolei w wyniku odjęcia przesunięcia otrzymamy wartość mniejszą od 65 (np. 63), to do tego wyniku należy dodać 26, aby otrzymać rozkodowany znak (63 + 26 = 89, czyli litera Y).

***Specyfikacja algorytmu.***

Program zaszyfowuje podany tekst zmodyfikowanym szyfrem Cezara.

*Dane wejściowe :*
tekst - łańcuch znaków zawierający rozszyfrowany tekst
klucz - przesunięcie w alfabecie, od której zaczynamy zastępować tekst, klucz $\in$ N,

*Dane wyjściowe :*

szyfr - łańcuch znaków zawierający zaszyfrowany tekst

*Dane pomocnicze :*
i - zmienna licznikowa
kod_ascii - przechowuje wartość ASCII danego znaku

***Lista kroków :***

Krok 1 : Podaj tekst; Podaj klucz;
Krok 2 : Dla i := 1 do konca długosci tekstu wykonuj kroki 3 .....9
Krok 3 : znak := tekst [i]; zamień małą literę na dużą
Krok 4 : Jeśli znak jest zawarty w zbiorze [‘A’...’Z’] wykonaj krok 5,6,7, 8, w przeciwnym wypadku przejdź do kroku 9
Krok 5 : kod_ascii := Ord (znak) + klucz; { zamień znak na kod ASCII , dodaj do kodu wartość klucza i podstaw pod zmienną kod_ascii }
Krok 6 : Jeśli kod_ascii > 90 to kod_ascii := kod_ascii – 26;
Krok 7 : Jeśli kod_ascii < 65 to kod_ascii := kod_ascii + 26;
Krok 8 : szyfr [i] := Chr (kod_ascii); { zamień kod ASCII na znak i podstaw do zmiennej szyfr }
Krok 9 : szyfr [i] := znak;
Krok 10 : Zakończ algorytm;

Jako alfabet szyfrowy można wybrać jakiekolwiek uporządkowanie liter alfabetu. Takich uporządkowań jest 35! (dla alfabetu polskiego) , co jest olbrzymią liczbą. Niepowołana osoba ma więc małe szanse natrafić przypadkowo na wybrany z tej liczby możliwości alfabet szyfrowy. Jednak wybór losowego uporządkowania liter w alfabecie szyfrowym ma wadę: alfabet ten trzeba w jakiś tajny sposób przekazać osobie, która ma odczytywać tworzone nim kryptogramy. Dlatego zamiast losowego wyboru alfabetu szyfrowego stosuje się różne rodzaje kluczy, które precyzyjnie określają alfabet szyfrowy.

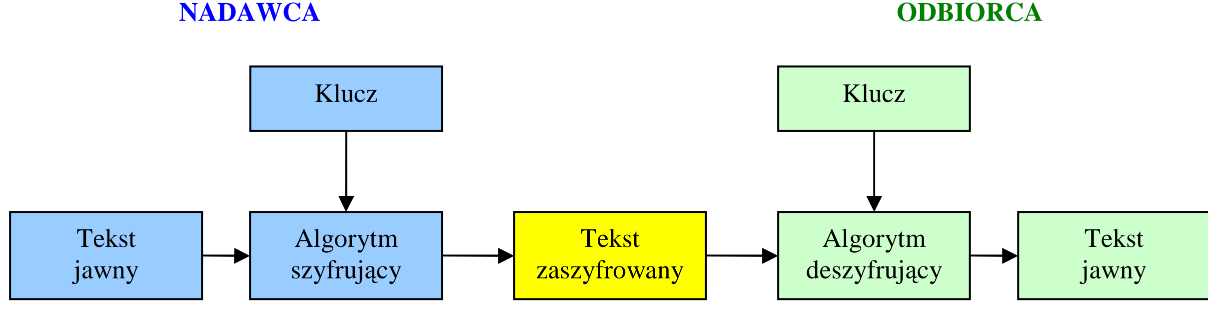
Jednym ze sposobów określania alfabetu szyfrowego jest podanie słowa (lub powiedzenia) kluczowego, na podstawie którego tworzy się alfabet szyfrowy. Dla przykładu, użyjemy słowa Cappadocia jako słowa kluczowego do utworzenia alfabetu szyfrowego. Najpierw umieszczamy słowo kluczowe na początku alfabetu szyfrowego i usuwamy z niego powtarzające się litery -w naszym przykładzie pozostaje więc CAPDOI. Następnie, usuwamy z alfabetu jawnego litery, które znajdują się słowie kluczowym (zaznaczone na czerwono) i dopisujemy pozostałe litery w porządku alfabetycznym, zaczynając od litery następnej po ostatniej literze w pozostałej części słowa kluczowego (w przykładzie od litery J) i kontynuując od początku alfabetu (w przykładzie poniżej oznaczono te miejsca pionową niebieską kreską ). Otrzymujemy:

| alfabet jawny | **a** | ą | b | **c** | ć | **d** | e | ę | f | g | h | **i** | j | k | l | ł | m | n | ń | **o** | ó | **p** | q | r | s | ś | t | u | v | w | x | y | z | ź | ż |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| alfabet szyfrowy | c | a | p | d | o | i | j | k | l | ł | m | n | ń | ó | q | r | s | ś | t | u | v | w | x | y | z | ź | ż | ą | b | ć | e | ę | f | g | h |

W czasie II Wojny Światowej w polskim ruchu oporu stosowano alfabety szyfrowe, określane na podstawie ustalonych fragmentów książek, które posiadali nadawca i odbiorca wiadomości.

Tworzenie alfabetów szyfrowych na podstawie słów kluczowych znacznie utrudnia deszyfrację kryptogramów, gdyż możliwych jest bardzo wiele słów (a ogólniej tekstów) kluczowych.

Przykładowe metody szyfrowania i deszyfrowania wiadomości, przedstawione powyżej, można przedstawić za pomocą schematu :




Bezpieczeństwo takiego sposobu przekazywania utajnionych wiadomości zależy głównie od utajnienia klucza. Osoba, która przechwyci tak utajnioną wiadomość, wiedząc nawet, jaki algorytm został użyty do jej zaszyfrowania, bez znajomości użytego klucza szyfrowania nie powinna móc ani odszyfrować kryptogramu, ani tym bardziej określić klucza szyfrowania.

Istnieje jednak kryptoanaliza, która zajmuje się deszyfrowaniem tekstów zaszyfrowanych bez znajomości klucza. Kryptografia i kryptoanaliza rywalizują ze sobą od początków utajniania wiadomości - kryptografia tworzy coraz doskonalsze (bezpieczniejsze) szyfry, a kryptoalnaliza dostarcza metod ich łamania.

Złamanie uogólnionego szyfru Cezara jest dość proste, należy jedynie określić o ile pozycji przesuwają się litery. Podobnie jest ze złamaniem szyfru płotowego - należy określić jak wysoki jest płot, czyli ile ma rzędów.

A czy można złamać szyfr podstawieniowy z dowolnym alfabetem szyfrowania, generowanym np.przez słowo kluczowe?

Jeśli tak, to na pewno nie jest to metoda sprawdzająca wszystkie możliwe alfabety szyfrowania, gdyż jest ich zbyt dużo. Pierwszy zapis o skutecznym sposobie łamania szyfru podstawieniowego z alfabetem szyfrowania pochodzi z IX wieku. Jego autorem był Al-Kindi, zwany „filozofem Arabów". Jako pierwszy wykorzystał on różnice w częstości występowania liter w tekstach (patrz tablica częstotliwości w rozdziale 14.4).

Oczywiście odnosi się to do nieco dłuższych tekstów; w krótszych kryptogramach najczęściej występująca w nich litera może odpowiadać którejś z następnych co do częstości liter. Ponadto, szukając par odpowiadających sobie liter uwzględnia się charakterystyczne dla danego języka połączenia dwóch lub więcej liter, np. w języku polskim dość często występują: rz, sz, cz, śś, a w języku angielskim - qu, th. Sposób łamania szyfru, czyli deszyfracji wiadomości, polegający na wykorzystaniu częstości występowania liter i ich kombinacji w tekście nazywa się analizą częstości.

Posłużenie się analizą częstości przy deszyfrowaniu tekstu nie jest tylko prostym skorzystaniem z tabeli częstości liter, ale żmudną analizą możliwych przypadków, w której nierzadko trzeba postępować metodą prób i błędów. Procesu deszyfracji, wykorzystującego analizę częstości, nie daje się prosto zautomatyzować za pomocą komputera. Jest to postępowanie w dużym stopniu interaktywne.

Po złamaniu prostego szyfru podstawieniowego, czyli z jednym alfabetem szyfrowym, nastąpił ruch kryptografów, którzy zaproponowali szyfr z wieloma alfabetami szyfrowymi, tzw. **szyfr polialfabetyczny**. Taki szyfr uniemożliwia użycie prostej analizy częstości, gdyż w tworzonych kryptogramach za daną literę jest podstawianych wiele różnych liter. Słowo kluczowe w tym przypadku służy do określenia alfabetów - są nimi alfabety Cezara wyznaczone przez kolejne jego litery. Dla przykładu, niech słowem kluczowym będzie BRIDE (dla utrudnienia kryptoanalizy, słowa kluczowe są wybierane z innych języków), wtedy mamy pięć alfabetów szyfrowych, zaczynające się od liter B, R, I, D i E:

| **alfabet jawny** | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| alfabet szyfrowy | **b** | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | a |
| alfabet szyfrowy | **r** | s | t | u | v | w | x | y | z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
| alfabet szyfrowy | **i** | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | a | b | c | d | e | f | g | h |
| alfabet szyfrowy | **d** | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | a | b | c |
| alfabet szyfrowy | **e** | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | a | b | c | d |

Zaszyfrujmy teraz wiadomość „*panna w opalach*" : pierwszą literę jawnego tekstu szyfrujemy posługując się pierwszym alfabetem szyfrowym, drugą - drugim, trzecią - trzecim, czwartą - czwartym, piątą - piątym, szóstą - pierwszym itd. Otrzymujemy więc kryptogram: „*QRVQE X FXDPBTP* " , w którym każda z powtarzających się liter w tekście jawnym (a i n) jest za każdym razem szyfrowana przez inną literę.

Taki sposób szyfrowania został zaproponowany w XVI wieku przez dyplomatę francuskiego Blaise de Vigenere'a i nazywa się **szyfrem Vigenere'a**.

Innym sposobem implementacji szyfru Vigenere'a jest użycie tzw. tablicy Vigenere'a (poniżej). *Uwaga: oznaczenia cyfrowe wierszy zastosowano tylko w celach edukacyjnych. Kolorem żółtym zaznaczono tekst jawny.*

| **0** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A |
| 2 | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B |
| 3 | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |
| 4 | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D |
| 5 | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
| 6 | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F |
| 7 | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G |
| 8 | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H |
| 9 | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I |
| 10 | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J |
| 11 | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K |
| 12 | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L |
| 13 | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M |
| 14 | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| 15 | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 16 | Q | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
| 17 | R | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| 18 | S | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
| 19 | T | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| 20 | U | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| 21 | V | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| 22 | W | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V |
| 23 | X | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
| 24 | Y | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X |
| 25 | Z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |
| 26 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

Zwróćmy uwagę, że tablica ta powstała w wyniku przesunięcia alfabetu szyfrowego o jedną pozycję w lewo (patrz : szyfr Cezara). W szyfrze tym kolejne litery listu szyfrujemy, posługując się różnymi wierszami tablicy (różnymi alfabetami). Inaczej mówiąc, nadawca może zaszyfrować pierwszą literę, używając 5 wiersza, drugą literę – używając 14, trzecią - używając 21, i tak dalej. W celu odczytania listu adresat musi wiedzieć, który wiersz tablicy został użyty do zaszyfrowanej każdej litery, a zatem musi istnieć uzgodniony system zmiany wierszy. Można to osiągnąć za pomocą słowa kluczowego.

Zaszyfrujmy więc teraz wiadomość „*panna w opalach*" używając klucza „BRIDGE". W tym celu możemy stworzyć sobie dodatkową tabelę (poniżej), w którą wpiszemy słowo kluczowe (wiersz 1) . Wpisujemy je tyle razy aby pokryło się z szyfrowaną wiadomością.

W drugim wierszu wpisujemy treść tekstu jawnego (pomijając spacje) i zaczynamy szyfrowanie.

| Klucz | B | R | I | D | E | B | R | I | D | E | B | R | I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tekst jawny | P | A | N | N | A | W | O | P | A | L | A | C | H |
| tekst tajny | q | r | v | q | e | x | f | x | d | p | b | t | p |

Szyfrujemy pierwszą literę tekstu (P). Literze P odpowiada w kluczu litera B. Litera ta (B) określa numer wiersza w tablicy : 1. Teraz szukamy jaka litera z wiersza 1 odpowiada literze P (z wiersza 0 ) . Jest nią litera Q . Teraz szyfrujemy literę A. Wybieramy wiersz 17 i szukamy w tym wierszu odpowiednika litery A – jest nią R. Trzecia litera to N – wiersz 8, odpowiednik V itd.

Alfabet szyfru ma postać : $F_i(a) = (a + k_i \mod N) \mod N$
gdzie :
$k_i$ – kolejna (i-ta) litera szyfru (dla powyższego przykładu : $k_1$ = B, $k_2$ = R ..., $k_6$ = E)
i – numer alfabetu (wiersz z tablicy)
N – długość alfabetu ( tu : 26 )
a – pozycja litery w alfabecie.

Oczywiście jest to szyfr symetryczny, więc funkcja deszyfrująca będzie miała postać : $G_i(x) = (x - k_i \mod N) \mod N$ gdzie x : pozycja zaszyfrowanej litery w alfabecie.

***Specyfikacja algorytmu.***

Program zaszyfowuje podany tekst szyfrem Vigenere'a.

*Dane wejściowe :*
tekst - łańcuch znaków zawierający rozszyfrowany tekst
klucz - łancuch zawierający słowo(a) – klucz szyfrujący

*Dane wyjściowe :*

szyfr - łańcuch znaków zawierający zaszyfrowany tekst

*Dane pomocnicze :*
i,j - zmienne licznikowe
poz - pozycja zaszyfrowanej litery w tablicy alfabet
alfabet - tablica typu znakowego przechowująca alfabet
długosc_alfabetu – stała informująca o długości alfabetu (dla naszego przykładu = 26 )

***Lista kroków :***

Krok 1 : Podaj tekst; Podaj klucz;
Krok 2 : Stwórz tablicę alfabetu w taki sposób by na pozycji 0 znajdowała się litera A, na pozycji 2 – B , 3 – C itd.
Krok 3 : Usuń spacje ze zmiennej tekst oraz ze zmiennej klucz oraz zamień w tych zmiennych wszystkie małe litery na duże.
Krok 4 : j := 1; szyfr := '';
Krok 5 : Dla i = 1, 2 , 3 ........długość_tekstu_do_zaszyfrowania wykonuj kroki 6 .... 9
Krok 6 : pobierz i-tą literę tekstu i zamień ją na liczbę (jej kod ASCII)
pobierz j-tą literę klucza i zamień ją na liczbę (jej kod ASCII) i wyznacz resztę z dzielenia przez długość alfabetu

| | |
|---|---|
| Krok 7 : | Zsumuj uzyskane w kroku 6 liczby. Sumę tą podziel przez długość alfabetu (stała dlugosc_alfabetu). Uzyskana reszta z tego dzielenia będzie wyznaczała pozycję zaszyfrowanej litery w alfabecie. |
| Krok 8 : | Odczytaj literę z obliczonej w kroku 7 pozycji i dodaj ją do zmiennej szyfr (na koniec) |
| Krok 9 : | Zwiększ licznik j o 1. Jeśli j > od długości klucza to j := 1 |
| Krok 10 : | Wypisz zaszyfrowany tekst (zmienna szyfr); |
| Krok 11 : | Zakończ algorytm; |

**Zadanie :** Odszyfrować tekst zapisany w pliku szyfr_1.txt. Tekst ten jest zaszyfrowany za pomocą szyfru Vigenere'a. Klucz : „To be or not to be". Wynik zapisać do pliku „odszyfr1.txt". (opieramy się na lafabecie 26-cio literowym).

*Tekst do odszyfrowania :*

WSTDMWEOVCOKILHPTSINQCTCEAKCUROUBGSRTSSPOOMO
JMMYKCXEGWFSHRXWXCKMELBPWQZAOSRKBQRGACTIBKTG
WFQSYMYQQRALRAFXKMDDBPZUWRNARTXPQCJPYENQAEGO
EEKTLKRLHBVVNBHCFXFXLZFRBOLPITMOHWPJSKQZRWVF
HXMMUEDNBWNPSFHPOOCMSLWHDBNXKCAWSMGVCNNQHLCC
EGWFYDINKGBCOEICQVNVPVPRQFRBIAEGQLTKHKBRXXXM
OUBAHLQJABSEDOTGMEDCAIGOEEKTNNTLHPWHKBPGQLTK
VSAEKO

Dopiero w połowie XIX wieku kryptoanalitycy złamali szyfr Vigenere'a -jednym z ich był Charles Babbage. W odpowiedzi na złamanie szyfru Vigenere'a, kryptografowie zaproponowali jednoczesne szyfrowanie par liter - **szyfr Playfair**.

Szyfr Playfair nie koduje pojedyńczych liter, lecz grupy dwóch znaków jednocześnie. Szyfr ten opiera się na kluczu, względem którego tworzona jest 25 elementowa dwuwymiarowa tablica znaków.

Ustalamy najpierw słowo kluczowe np. CHARLES. Przed rozpoczęciem szyfrowania zapisujemy litery alfabetu w postaci kwadratu 5x5, zaczynając od słowa kluczowego i uzupełniając resztę liter w kolejności alfabetycznej, bez powtarzania liter występujących w haśle oraz łącząc litery I i J. Nasza tablica będzie miała postać :

| | | | | |
|---|---|---|---|---|
| **C** | **H** | **A** | **R** | **L** |
| **E** | **S** | B | D | F |
| G | I/J | K | M | N |
| O | P | Q | T | U |
| V | W | X | Y | Z |

Jeśli w słowie kluczowym powielają się litery to opuszczamy następne powielające się. Ponieważ w algorytmie szyfrowania Playfair litery J oraz I są równoznaczne to możemy zamiast J pisać I. Na przykład słowo kluczowe TAJEMNICA, należałoby wpisać jako TAIEMNC.

Następnie dzielimy tekst jawny na pary liter, czyli ***digramy***. Każda para powinna składać się z dwóch różnych liter, co można osiągnąć wstawiając w razie konieczności dodatkową literę X. W przypadku, gdy mamy nieparzystą liczbę liter, na końcu dopisujemy także literę X.

Zakodujmy tekst : meet me at hammersmith bridge tonight.
Tekst jawny w postaci digramów będzie miał postać :
ME – ET – ME – AT – HA – M**X** – ME – RS – MI – TH – BR – ID – GE – TO – NI – GH – T**X**

Teraz można już zacząć proces szyfrowania. Wszystkie digramy można podzielić na trzy kategorie:

- obie litery znajdują się w tym samym wierszu,
- obie litery znajdują się w tej samej kolumnie
- wszystkie pozostałe.

Jeśli obie litery digramu znajdują się w tym samym wierszu, zastępujemy je sąsiednimi literami z prawej strony; tak więc MI zmienia się w NK.

| | | | | |
|---|---|---|---|---|
| C | H | A | R | L |
| E | S | B | D | F |
| G | **I** | **K** | **M** | **N** |
| O | P | Q | T | U |
| V | W | X | Y | Z |

Jeśli obie litery digramu znajdują się w tym samym wierszu ale jedna z liter digramu znajduje się na końcu wiersza, zastępujemy ją pierwszą literą wiersza, zatem NI zamienia się w GK.

| | | | | |
|---|---|---|---|---|
| C | H | A | R | L |
| E | S | B | D | F |
| **G** | **I** | **K** | M | **N** |
| O | P | Q | T | U |
| V | W | X | Y | Z |

Jeśli obie litery znajdują się w tej samej kolumnie, zastępujemy je literami leżącymi bezpośrednio pod nimi; tak więc EG zmienia się w GO.

| | | | | |
|---|---|---|---|---|
| C | H | A | R | L |
| **E** | S | B | D | F |
| **G/G** | I | K | M | N |
| **O** | P | Q | T | U |
| V | W | X | Y | Z |

Jeśli obie litery znajdują się w tej samej kolumnie i jedna z liter znajduje się na samym dole kolumny, zastępujemy ją pierwszą literą w kolumnie, na przykład EV zmienia się GC.

| | | | | |
|---|---|---|---|---|
| **C** | H | A | R | L |
| **E** | S | B | D | F |
| **G** | I | K | M | N |
| O | P | Q | T | U |
| **V** | W | X | Y | Z |

Jeśli litery digramu nie znajdują się ani w tym samym wierszu, ani w tej samej kolumnie, obowiązuje inna reguła. Aby zaszyfrować pierwszą literę idziemy wzdłuż wiersza, aż dotrzemy do kolumny zawierającej drugą literę, litera na skrzyżowaniu wiersza i kolumny reprezentuje pierwszą literę. W celu zaszyfrowania drugiej litery idziemy wzdłuż wiersza, aż dotrzemy do kolumny zawierającej pierwszą literę; litera na skrzyżowaniu zastępuje drugą literę digramu. Zatem ME zmienia się w GD,

C H A R L
**E** S B **D** F
**G** I K **M** N
O P Q T U
V W X Y Z

***Uwaga : nie można zmieniać dowolnie kolejności liter ! Digram CS jest różny od SC !***
Po zaszyfrowaniu nasza wiadomość ma postać:

| tekst jawny | ME | ET | ME | AT | HA | MX | ME | RS | MI | TH | BR | ID | GE | TO | NI | GH | TX |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tekst tajny | GD | DO | GD | RQ | AR | KY | GD | HD | NK | PR | DA | MS | OG | UP | GK | IC | QY |

Oczywiście na etapie ustalania klucza, należy wybrać klucz stosunkowo długi. Proces deszyfrowania przebiega w sposów identyczny jak w przypadku szyfrowania - jest to klasyczny przykład szyfru symetrycznego. Jest to prosty monoalfabetyczny szyfr podstawieniowy i można go złamać, korzystając z analizy częstości.

W latach dwudziestych pojawiła się pierwsza maszyna szyfrująca - Enigma - zastąpiła ona dotychczas „obowiązujący" algorytm szyfrujący i deszyfrujący, a kluczem w tej maszynie było początkowe ustawienie jej elementów mechanicznych. Na początku maszynę tę można było nawet kupić na wolnym rynku. Szyfr Enigmy został złamany przez zespół polskich matematyków, którym kierował inż. Marian Rejewski. Zbudowali oni maszynę deszyfrującą, zwaną Bombą, która składała się z wielu kopii Enigmy. Swoje pomysły, wraz z maszyną, Polacy przekazali jeszcze przed wybuchem wojny Brytyjczykom, którzy w Blatchlay Park koło Londynu zapoczątkowali erę komputerowej kryptografii i kryptoanalizy.

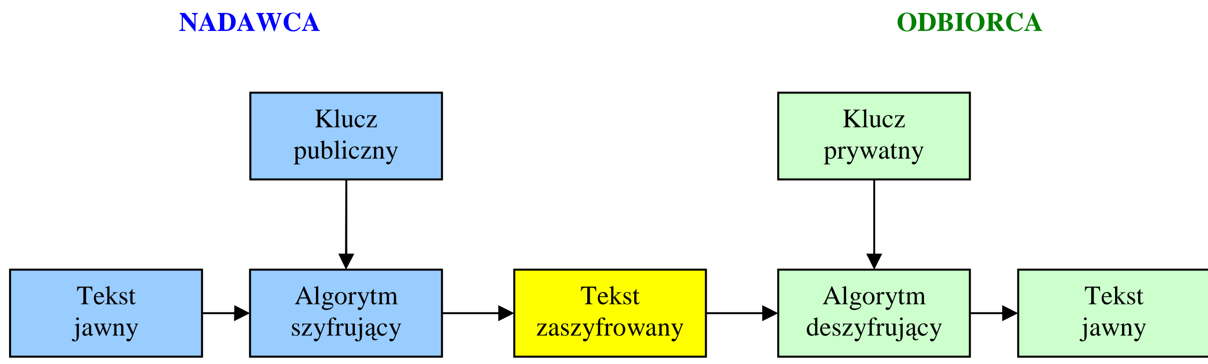
## *14.5. Kryptografia z kluczem jawnym.*

Pojawianie się coraz silniejszych komputerów powoduje realne zagrożenie dla przesyłania utajnionych wiadomości. Kryptoanalityk może skorzystać z mocy komputera, by sprawdzić bardzo dużą liczbę możliwych alfabetów i kluczy oraz prowadzić analizę kryptogramów metodą prób i błędów. Co więcej, z ekspansją komunikacji najpierw telefonicznej, a obecnie - internetowej wzrosła do olbrzymich rozmiarów liczba przesyłanych wiadomości. Państwa, instytucje, a także pojedynczy obywatele chcieliby mieć gwarancję, że system wymiany wiadomości może zapewnić im bezpieczeństwo i prywatność komunikacji.

W 1976 roku przyjęto w USA **szyfr Lucifer** jako standard komputerowego szyfrowania danych DES (ang. Data Encryption Standard). Liczba możliwych kluczy dla systemu DES jest gwarancją, że praktycznie jest to bezpieczny szyfr - powszechnie dostępne komputery są zbyt słabe,

by go złamać. Pozostaje jednak nadal nierozwiązany tzw. problem dystrybucji klucza - w jaki sposób dostarczyć klucz odbiorcy utajnionej nim wiadomości. Problem ten dotyczył większości szyfrów w historii kryptografii. Na przykład Marian Rejewski osiągnął pierwsze sukcesy przy deszyfracji Enigmy, korzystając min. z faktu, że klucz do zaszyfrowanej wiadomości był dwa razy powtarzany na początku wiadomości.

W połowie lat siedemdziesiątych pojawiła się sugestia, że wymiana klucza między komunikującymi się stronami być może nie jest konieczna. Tak zrodził się pomysł szyfru z jawnym kluczem, którego schemat przedstawiony jest poniżej.




Schemat ten różni się od poprzedniego schematu tym, że zamiast jednego klucza dla nadawcy i odbiorcy mamy parę kluczy: klucz publiczny, zwanym kluczem jawnym, i klucz prywatny, zwany też kluczem tajnym. Jest to więc szyfr asymetryczny. Działanie odbiorcy i nadawcy utajnionych wiadomości w tym przypadku jest następujące:

- Odbiorca wiadomości tworzy parę kluczy: publiczny i prywatny i ujawnia klucz publiczny, np. zamieszcza go na swojej stronie internetowej.
- Ktokolwiek chce wysłać zaszyfrowaną wiadomość do odbiorcy szyfruje ją jego kluczem publicznym, zaś tak utworzony kryptogram można odczytać jedynie posługując się kluczem prywatnym odbiorcy.

Para kluczy - publiczny i prywatny - ma jeszcze tę własność, że znajomość klucza publicznego nie wystarcza nie tylko do odszyfrowania wiadomości nim zaszyfrowanej, ale również nie umożliwia utworzenia klucza prywatnego, który jest niezbędny do odszyfrowania wiadomości. Utworzenie takiej pary kluczy było możliwe dopiero w erze komputerów. Odbiorca może łatwo określić oba klucze z pary, ale nikt poza nim, przy obecnym stanie wiedzy i mocy komputerów nie jest w stanie odtworzyć klucza prywatnego na podstawie klucza publicznego.

Tak naprawdę bezpieczeństwo szyfrowania z kluczem publicznym wynika z niedoskonałości komputerów ☺ , które nie są w stanie znaleźć rozwiązania dosyć prostego, ale na bardzo dużych liczbach.

Pierwszy szyfr z kluczem jawnym opracowali w 1977 roku Ronald **R**ivest, Adi **S**hamir i Leonard **A**dleman z MIT i od inicjałów ich nazwisk pochodzi jego nazwa szyfr RSA.

### *14.5.1. Szyfr RSA.*

Szyfr ten oparty na kluczu asymetrycznym. Może być wykorzystywany zarówno do szyfrowania jak i elektronicznego podpisywania dokumentów. Prawa do algorytmu RSA posiada firma RSA Security Inc., udzielająca płatnej licencji na jego używanie w programach innych producentów (patent ważny jest jednak tylko w USA). RSA wkomponowany jest m.in. w dwie najważniejsze przeglądarki: Netscape Navigator oraz Internet Explorer.

Fundamentem RSA jest algorytm służący do generowania unikalnych i bezpiecznych (odpornych na próby odgadnięcia) par kluczy. Mnoży on dwie duże liczby pierwsze i z otrzymanego wyniku poprzez kilka innych dodatkowych operacji ustala klucz publiczny i zależny od niego klucz prywatny. Pierwszy z nich służy do szyfrowania wiadomości przeznaczonych dla właściciela kluczy i co za tym idzie powinien być jak najszerzej propagowany. Klucz prywatny jest tajny i tylko przy jego pomocy można odszyfrować to, co zostało zakodowane kluczem publicznym.

Algorytm RSA bazuje na dwóch dostatecznie dużych liczbach pierwszych. Na podstawie tych liczb (oznaczonych dalej jako p i q) obliczane są dwie kolejne wartości: n oraz e, z których:

$n = p * q$ i $e < n$

gdzie e jest liczbą względnie pierwszą[1] do iloczynu $( p - 1 )*( q - 1 )$

Ostatni etap to wyznaczenie takiej liczby d, że:

$e * d$ jest podzielne przez $( p - 1 )*( q - 1 )$

Liczbę n nazywamy modułem RSA, liczbę e nazywamy wyładnikiem szyfrującym, zaś liczbę d - wykładnikiem deszyfryjącym.

Od tego momentu liczby pierwsze p i q nie biorą udziału w obliczeniach i mogą zostać zniszczone. W dalszych operacjach korzysta się już z wyznaczonego klucza publicznego (para liczb n oraz e) i klucza prywatnego (para liczb n oraz d). Chcąc uzyskać zaszyfrowaną postać „s" wiadomości tekstowej w należy przeprowadzić następującą operację dzielenia modulo (dzielenie z resztą ) z użyciem klucza publicznego:

$$s = w^{e} \bmod n$$

Deszyfrowanie odbywa się w podobny sposób, lecz tym razem w operacji bierze udział klucz prywatny:

$$w = s^{d} \bmod n$$

Bezpieczeństwo szyfru RSA opiera się na trudności rozkładu dużej liczby „n" na czynniki. Nie jest znany żaden efektywny algorytm rozkładu dowolnej takiej liczby działający w rozsądnie krótkim czasie. Oczywiście, dla pewnych klas liczb naturalnych istnieją algorytmy umożliwiające ich rozkład na czynniki w krótkim czasie. Dlatego wymaga się, aby liczby pierwsze p, q wykorzystywane do konstrukcji klucza spełniały pewne postulaty.
Aktualnie przyjmuje się, że należy dokonać wyboru p i q tak, aby spełnione były następujące warunki :

[1] Liczby względnie pierwsze – dwie liczby całkowite, których największym wspólnym podzielnikiem jest liczba 1.

- obie liczby p i q są wybrane losowo;
- obie liczby p i q mają co najmniej po 150 cyfr dziesiętnych;
- jedna z nich jest o kilka rzędów wielkości mniejsza od drugiej, ale, jednocześnie, ich różnica jest względnie mała;
- (p - 1) / 2 i (q - 1) / 2 są liczbami pierwszymi;
- p + 1 oraz q + 1 mają duże czynniki pierwsze p' i q' odpowiednio;
- NWD(p - 1; q - 1) jest niewielki.

Uważa się, że liczby spełniające powyższe postulaty są tzw. mocnymi liczbami pierwszymi, czyli liczbami pierwszymi, których iloczyn jest wyjątkowo trudny do faktoryzacji (rozkładu na czynniki pierwsze) za pomocą znanych aktualnie algorytmów.

Poważną wadą algorytmu RSA jest jego wolne działanie. Z tego powodu stosuje się go zazwyczaj w połączeniu z innymi algorytmami, np. DES, który operacje szyfrowania przeprowadza 1000 razy szybciej. W takich systemach hybrydowych DES służy do szyfrowania wiadomości, RSA natomiast koduje już tylko klucz używany w DES-ie. Klucz zamknięty w takiej elektronicznej kopercie może następnie zostać bezpiecznie przesłany kanałem nie zapewniającym poufności - np. przez Internet.

RSA wchodzi w skład wielu istniejących lub proponowanych standardów i protokołów sieciowych. Jest szeroko stosowany w komunikacji internetowej: poufnej poczcie elektronicznej i sygnowaniu dokumentów elektronicznymi podpisami, systemie PGP i protokołach SET, S/MIME, SSL oraz HTTPS.

PRZYKŁAD :

W opisie szyfrowania i deszyfrowania metodą RSA, posłużymy się imionami odbiorcy - Alicja i nadawcy - Bogdan. Dla ułatwienia przyjmiemy małe liczby pierwsze.

*Etap przygotowania kluczy*

- Alicja wybiera dwie leżące bliskie sobie liczby pierwsze p i q np. p=11 i q=13. Liczby te Alicja trzyma w tajemnicy.
- Teraz Alicja oblicza n =p*q i wybiera dowolną liczbę naturalną e, która jest względnie pierwsza z liczbą (p - l)*(q - 1), czyli te dwie liczby nie mają wspólnych dzielników poza liczbą 1.
  W naszym przypadku mamy **n** = 11*13 = **143**.
  Ponieważ (p - l)*(q - 1) = 10*12 = 120 = $2^3 * 3 * 5$, więc możemy przyjąć **e = 7**.
- Alicja znajduje liczbę naturalną d taką, że e*d = 1 (mod (p - l)*(q - 1)), czyli reszta z dzielenia e*d przez (p - l)*(q - 1) jest równa 1. Liczbę d można znaleźć, posługując się algorytmem Euklidesa, o którym była mowa we wcześniejszych rozdziałach tego opracowania.
  Czyli w naszym przypadku otrzymujemy :
  7*d = 1 (mod 120). Taką liczbą jest np. **d = 103**, gdyż 7*103 = 721 i 721 podzielone przez 120 daje resztę 1.
- Alicja ogłasza parę liczb (n, e) jako swój klucz publiczny, np. w Internecie, a parę (n, d) zachowuje w tajemnicy jako klucz prywatny. Jednocześnie niszczy liczby p i q, by nie wpadły w niczyje ręce, co mogłoby umożliwić odtworzenie jej klucza prywatnego. Jeśli p i q są dostatecznie dużymi liczbami pierwszymi, to znajomość „n" i „e" nie wystarcza, by obliczyć wartość d posługując się nawet najpotężniejszymi dzisiaj komputerami.

*Szyfrowanie wiadomości*

- Jeśli Bogdan chce wysłać Alicji jakąś wiadomość, to najpierw musi ją przedstawić w postaci liczby naturalnej M, nie większej niż n. Tekst można zamienić na liczbę posługując się kodem ASCII. Jeśli wiadomość jest zbyt długa, to należy ją szyfrować blokami odpowiedniej wielkości.
  Przykładowo, jako wiadomość wybierzmy literę Q, która w kodzie ASCII ma kod 81. Przyjmujemy więc za wiadomość do wysłania M = 81.
- Wiadomość, jaką Bogdan wysyła do Alicji, jest liczbą : $P = M^e \bmod n$. Obliczenie wartości P może wydawać bardzo złożone, ale można je szybko wykonać przy wykorzystaniu jednej z szybkich metod potęgowania (opisanych we wcześniejszych rozdziałach) oraz własności operacji na resztach.
  Przykład. Mamy obliczyć $P = 81^7 \bmod 143$. Korzystając z rozkładu binarnego liczby 7, mamy $81^7 = 81^1 * 81^2 * 81^4$ . Z każdej z tych potęg obliczamy tylko resztę z dzielenia przez 143. Otrzymujemy stąd: $(81^1 * 81^2 * 81^4) \bmod 143 = 81*126*3 = 16$ .
- Bogdan wysyła do Alicji wiadomość P = 16.

*Odszyfrowanie kryptogramu*

Alicja otrzymała od Bogdana zaszyfrowaną wiadomość P i aby ją odszyfrować oblicza $M = P^d \bmod n$.
Czyli : $M = 16^{103} \bmod 143$. Postępujemy podobnie, jak w punkcie 2 przy szyfrowaniu wiadomości. Mamy $16^{103} = 16^1 * 16^2 * 16^4 * 16^{32} * 16^{64}$ i z każdej z tych potęg obliczamy resztę z dzielenia przez 143.
Otrzymujemy: $(16^1 * 16^2 * 16^4 * 16^{32} * 16^{64}) \bmod 143 = (16 * 113 * 42 *113 * 42) \bmod 143 = 81$. Alicja wie, że Bogdan chciał jej przekazać w tajemnicy wiadomość brzmiącą: Q.

***Przykładowy program.***

```
{
 ******************************************************
 ** Przykladowa aplikacja obrazujaca spos˘b dzialania **
 ** asymetrycznego systemu kodowania informacji RSA.  **
 ** ------------------------------------------------- **
 **          (C)2003 mgr Jerzy Walaszek             **
 **           I Liceum Og˘lnokształcace            **
 **         im. Kazimierza Brodzinskiego            **
 **                w Tarnowie                    **
 ******************************************************
}

program rsa;
Uses Crt;

{ Procedura oczekuje na naciśnięcie klawisza Enter
 po czym czyści ekran okna konsoli
------------------------------------------------  }

procedure Czekaj;
var
  i : integer;
begin
  writeln;
```

```
  writeln('Zapisz te dane i nacisnij Enter');
  readln;
  for i := 1 to 500 do writeln;
end;

{ Funkcja obliczajaca NWD dla dw˘ch liczb
---------------------------------------  }

function nwd(a,b : integer) : integer;
var
  t : integer;
begin
  while b <> 0 do
  begin
    t := b;
    b := a mod b;
    a := t
  end;
  nwd := a
end;

{ Funkcja obliczania odwrotnosci modulo n
---------------------------------------    }

function odwr_mod(a,n : integer) : integer;
var
  a0,n0,p0,p1,q,r,t : integer;
begin
  p0 := 0; p1 := 1; a0 := a; n0 := n;
  q  := n0 div a0;
  r  := n0 mod a0;
  while r > 0 do
  begin
    t := p0 - q * p1;
    if t >= 0 then
      t := t mod n
    else
      t := n - ((-t) mod n);
    p0 := p1; p1 := t;
    n0 := a0; a0 := r;
    q  := n0 div a0;
    r  := n0 mod a0;
  end;
  odwr_mod := p1;
end;

{ Procedura generowania kluczy RSA
---------------------------------  }

procedure klucze_RSA;
const
  tp : array[0..9] of integer =
  (11,13,17,19,23,29,31,37,41,43);
var
  p,q,phi,n,e,d : integer;
begin
  writeln('Generowanie kluczy RSA');
  writeln('----------------------');
  writeln;

{ generujemy dwie rozne, losowe liczby pierwsze  }
```

```
  repeat
    p := tp[random(10)];
    q := tp[random(10)];
  until p <> q;

  phi := (p - 1) * (q - 1);
  n   := p * q;

{ wyznaczamy wykladniki e i d  }

  e := 3;
  while nwd(e,phi) <> 1 do inc(e,2);
  d := odwr_mod(e,phi);

{ gotowe, wypisujemy klucze  }

  writeln('KLUCZ PUBLICZNY');
  writeln('wykladnik e = ',e);
  writeln('    modul n = ',n);
  writeln;
  writeln('KLUCZ PRYWATNY');
  writeln('wykladnik d = ',d);
  Czekaj;
end;

{ Funkcja oblicza modulo potegi podanej liczby
--------------------------------------------- }

function pot_mod(a,w,n : integer) : integer;
var
  pot,wyn,q  : integer;
begin

{ wykladnik w rozkladamy na sume poteg 2. Dla reszt
{ niezerowych tworzymy iloczyn poteg a modulo n.    }

  pot := a; wyn := 1; q := w;
  while q > 0 do
  begin
    if (q mod 2) = 1 then wyn := (wyn * pot) mod n;
    pot := (pot * pot) mod n; { kolejna potega }
    q := q div 2;
  end;
  pot_mod := wyn;
end;

{ Procedura kodowania danych RSA
-------------------------------- }

procedure kodowanie_RSA;
var
  e,n,t : integer;
begin
  writeln('Kodowanie danych RSA');
  writeln('--------------------');
  writeln;
  write('Podaj wykladnik "e" = '); readln(e);
  write('    Podaj modul "n" = '); readln(n);
  writeln('----------------------------------');
  writeln;
  write('Podaj kod RSA (liczbe)  = '); readln(t);
  writeln;
```

```
  writeln('Wynik kodowania = ',pot_mod(t,e,n));
  Czekaj;
end;

{ *******************
 ** Program główny **
  ******************* }

var
  w : integer;

begin
  ClrScr;
  randomize;
  repeat
    writeln('System szyfrowania danych RSA');
    writeln('-----------------------------');
    writeln(' (C)2003  mgr Jerzy Walaszek ');
    writeln;
    writeln('MENU');
    writeln('====');
    writeln('[ 0 ] - Koniec pracy programu');
    writeln('[ 1 ] - Generowanie kluczy RSA');
    writeln('[ 2 ] - Kodowanie RSA');
    writeln;
    write('Jaki jest twoj wybor? (0, 1 lub 2) : ');
    readln(w);
    writeln; writeln; writeln;
    case w of
      1 : klucze_RSA;
      2 : kodowanie_RSA;
    end;
    writeln; writeln; writeln;
  until w = 0;
end.
```

## *14.6. Potwierdzenie autentyczności i podpis elektroniczny.*

Zapewne każdy uzytkownik systemu Windows XP spotkał się z komunikatem o sprawdzeniu autentyczności oprogramowania. W jaki sposób się to dzieje ? Odpowiedz daje algorytm RSA. Wytwórca koduje za pomocą klucza prywatnego informacje identyfikacyjne i umieszcza je w kodzie programu wraz z kluczem publicznym. Dzięki temu oprogramowanie użytkownika może rozszyfrować te dane i porównać je z listą autoryzowanych wytwórców. Zwróć uwagę, iż tylko autentyczny wytwórca może takie dane umieścić w kodzie, ponieważ tylko on posiada klucz, za pomocą którego dane te zostały zaszyfrowane. Klucz publiczny pozwala je jedynie rozszyfrować.

Oczywiście dane umieszczone w ten sposób w jakimś produkcie muszą go jednoznacznie opisywać, aby nie można było wyciąć zaszyfrowanego kodu i wkleić do innego produktu. Jednakże nie jest to już problem szyfrowania tylko implementacji systemu potwierdzania autentyczności.

Na tej samej zasadzie funkcjonuje tzw. podpis elektroniczny zwany również podpisem cyfrowym, czyli podpis na dokumentach elektronicznych, a raczej - podpis towarzyszący takim dokumentom. O jego znaczeniu może świadczyć fakt, że państwa Unii Europejskiej wprowadzają przepisy, uznające podpis elektroniczny za równorzędny z odręcznym. W Polsce również uchwalono w sejmie odpowiednią ustawę (16 sierpnia 2002 r.). Zatem w przyszłości, zamiast iść do

banku, by podpisać umowę o kredyt wystarczy taką umowę zaszyfrować, dołączyć do niej podpis elektroniczny i przesłać pocztą elektroniczną.

Podpis na dokumencie (tradycyjnym lub elektronicznym) powinien mieć następujące cechy:

- zapewnia jednoznaczną identyfikację autora - nikt inny nie posługuje się takim samym podpisem;
- nie można go podrobić;
- nie można go skopiować na inny dokument - mechanicznie lub elektronicznie;
- gwarantuje, że po podpisaniu nim dokumentu nikt nie może wprowadzić żadnych zmian do tego dokumentu.

Opiszemy teraz co to jest podpis elektroniczny i jak się nim posługiwać.
Sam podpis elektroniczny jest umownym ciągiem znaków, nierozerwalnie związanych z podpisywanym dokumentem. Osoba, która chce posługiwać się takim podpisem, musi najpierw skontaktować się z centrum certyfikacji. Tam otrzymuje parę kluczy - publiczny i prywatny oraz osobisty certyfikat elektroniczny, który będzie zawierał m.in. dane tej osoby i jej klucz publiczny. Przyznawane certyfikaty elektroniczne będą dostępne w Internecie, aby odbiorcy dokumentów mieli pewność, kto jest adresatem przesyłki.

Przy posługiwaniu się podpisem elektronicznym ważne jest pojęcie tzw. skrót dokumentu. Skrót dokumentu stanowi jego jednoznaczną reprezentację; jakakolwiek zmiana w treści dokumentu powoduje zmianę w jego skrócie. Skrót powstaje przez zastosowanie do dokumentu odpowiedniego algorytmu (będącego realizacją tzw. funkcji mieszającej). Po utworzeniu skrótu dokumentu, zostaje on zaszyfrowany i na stałe związany z dokumentem (również zaszyfrowanym), na podstawie którego powstał, i służy do identyfikacji podpisującego.

Teraz opiszemy, na czym polega podpis elektroniczny dokumentu. Przypuśćmy, że Pan A chce wysłać dokument do Pana B i podpisać go elektronicznie. Dokument znajduje się w komputerze, tam również jest dostępny certyfikat elektroniczny Pana A. Aby wysłać dokument z podpisem elektronicznym Pan A powinien nacisnąć przycisk Podpisz (mamy tu na myśli oprogramowanie, związane z podpisem elektronicznym). Wtedy są wykonywane następujące czynności:

- Tworzony jest skrót z dokumentu, przeznaczonego do wysyłki.
- Skrót ten zostaje zaszyfrowany kluczem prywatnym Pana A.
- Do dokumentu zostaje dołączony zaszyfrowany jego skrót i certyfikat Pana A z jego kluczem publicznym.

Tak podpisany dokument jest przesyłany do odbiorcy (Pana B), np. pocztą elektroniczną.

Odbiorca, po otrzymaniu dokumentu, podpisanego elektronicznie, aby zweryfikować jego autentyczność, naciska przycisk Weryfikuj. Wtedy:

- Tworzony jest skrót z otrzymanego dokumentu.
- Dołączony do dokumentu skrót zostaje rozszyfrowany kluczem publicznym Pana A, zawartym w jego certyfikacie elektronicznym.
- Skrót utworzony z otrzymanego dokumentu i odszyfrowany skrót zostają porównane. Jeśli są zgodne, to oznacza, że Pan A, czyli osoba widniejąca na certyfikacie, jest jego autorem.

Szczegółowy opis działania i tworzenia podpisu elektronicznego jest opisany na stronie pod adresem http://www.podpiselektroniczny.pl. Można tam również znaleźć informacje o centrach certyfikacji, jak również utworzyć swój osobisty certyfikat.